GemStone

GemStone Programming Guide

July 1996

GemStone

Version 5.0

IMPORTANT NOTICE

This manual and the information contained in it are furnished for informational use only and are subject to change without notice. GemStone Systems, Inc. assumes no responsibility or liability for any errors or inaccuracies that may appear in this manual or in the information contained in it. The manual, or any part of it, may not be reproduced, displayed, photocopied, transmitted or otherwise copied in any form or by any means now known or later developed, such as electronic, optical or mechanical means, without written authorization from GemStone Systems, Inc. Any unauthorized copying may be a violation of law.

The software installed in accordance with this manual is copyrighted and licensed by GemStone Systems, Inc. under separate license agreement. This software may only be used pursuant to the terms and conditions of such license agreement. Any other use may be a violation of law.

Limitations

The software described in this manual is a customer-supported product. Due to the customer's ability to change any part of a Smalltalk image, GemStone Systems, Inc. cannot guarantee that the GemStone programming environment will function with all Smalltalk images.

Copyright by GemStone Systems, Inc. 1988–1995. All rights reserved.

Use, duplication, or disclosure by the Government is subject to restrictions set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

Trademarks

GemStone is a registered trademark of GemStone Systems, Inc.

Objectworks and **Smalltalk-80** are trademarks of ParcPlace Systems, Inc.

Smalltalk/V is a registered trademark of Digitalk, Inc.

Sun, **Sun Microsystems**, **Solaris** and **SunOS** are trademarks or registered trademarks of Sun Microsystems, Inc. All **SPARC** trademarks, including **SPARCstation**, are trademarks or registered trademarks of SPARC International, Inc. **SPARCstation** is licensed exclusively to Sun Microsystems, Inc.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

Preface

About This Manual

This manual describes the GemStone Smalltalk language and programming environment — a bridge between your application's Smalltalk code running on a UNIX workstation and the GemStone database running on the host computer. Along with one of the interfaces for the programming environment, you can build comprehensive applications.

Intended Audience

This manual is intended for users familiar with the basic concepts of computer programming. It explains GemStone Smalltalk in terms of traditional programming concepts. Therefore, you'll benefit most from the material presented here if you have a solid understanding of a conventional language such as C.

It would also be helpful to be familiar with a Smalltalk language and its programming environment. In addition to your Smalltalk product manuals, we recommend *Smalltalk-80: The Language and its Implementation* and *Smalltalk-80: The Interactive Programming Environment* (both published by Addison-Wesley).

This manual assumes that the GemStone system has been correctly installed on your host computer as described in the *GemStone System Administration Guide*, and

that your system meets the requirements listed in the *Installation* section of the *Release Notes*.

How This Manual Is Organized

The *GemStone Programming Guide* is a narrative introduction to the major topics in GemStone Smalltalk programming. A companion volume, the *GemStone Kernel Reference*, lists, in alphabetical order, each of the classes supplied for your use in Smalltalk programming and describes their complete functionality. We recommend that you use the *GemStone Kernel Reference* as a reference when necessary.

Documentation Conventions

Smalltalk code is printed in a monospace font throughout this manual. It looks like this:

```
numericArray add: (myVariable + 1)
```

When the result of executing an example is shown, it is underlined:

```
numericArray at: 1
  12486
```

Executing the Examples

This manual includes a many examples. Because we cannot be certain which interface you are using, and because the interface affects the way you execute the examples, here are a few words about the mechanics of the situation may be useful here.

There are two simple ways to write and compile a method:

- If you are using the GemStone Smalltalk Interface, you can use the structured editing and execution facilities provided by a GemStone Browser or Workspace. A browser makes it easier to define classes and methods by presenting templates for these operations. Once you've filled out the templates, a browser internally builds and executes Smalltalk expressions to compile the classes and methods. A browser organizes your work and presents it in a pleasing and easily understood format.

  A workspace makes it easier to compile and execute fragments of Smalltalk code interactively, and see the results immediately using the GemStone *print it* command.

- You can also enter your Smalltalk method code through the Topaz version of the programming environment. Topaz requires a few extra commands to begin and end an example. To identify code as constituting a method, for instance, you'll add a couple of simple non-Smalltalk directives such as "METHOD:." These tell Topaz to treat the indicated text as a method to be compiled and installed in a class.

  This is in some ways less convenient than using the GemStone Browser to create methods, but it has one important advantage: method definitions in this format are easily represented and inspected on the printed page.

This manual presents examples in Topaz format, with Topaz commands presented in boldface type. Those commands probably need little explanation when you see them in context; however, you may need to turn to the Topaz user manual for instructions about entering and executing the text of the upcoming examples.

If you are using the GemStone Smalltalk Interface, you may instead choose to read the introductions to the browser and workspace, and then use those tools to enter the examples in this manual. The text of the examples themselves (excluding the boldface Topaz commands) is the same whichever way you choose to enter it.

Other Useful Documents

You will find it useful to look at documents that describe other components of the GemStone data management system:

- A complete description of the behavior of each GemStone Smalltalk kernel class is available in the *GemStone Kernel Reference*.
- The Topaz interface allows you to process data and move it between the GemStone system and a terminal or workstation, when your Smalltalk program needs to read terminal input or send data to a workstation for local processing and display. The GemBuilder for Smalltalk and GemBuilder for C interfaces provide function libraries to access the repository. Each interface is described in its own user manual.
- In addition, if you will be acting as a system administrator, or developing software for someone else who must play those roles, read the *GemStone System Administration Guide*.

Technical Support

GemStone provides several sources for product information and support. GemStone product manuals provide extensive documentation, and should always be your first source of information. GemStone Technical Support engineers will refer you to these documents when applicable. However, you may need to contact Technical Support for the following reasons:

- Your technical question is not answered in the documentation.
- You receive an error message that directs you to contact GemStone Technical Support.
- You want to report a bug.
- You want to submit a feature request.

Questions concerning product availability, pricing, keyfiles, or future features should be directed to your GemStone account manager.

When contacting GemStone Technical Support, please be prepared to provide the following information:

- Your name, company name, and GemStone license number,
- the GemStone product and version you are using,
- the hardware platform and operating system you are using,
- a description of the problem or request,
- exact error message(s) received, if any.

Your GemStone support agreement may identify specific individuals who are responsible for submitting all support requests to GemStone. If so, please submit your information through those individuals. All responses will be sent to authorized contacts only.

For non-emergency requests, you should contact Technical Support by email, Web form, or facsimile. You will receive confirmation of your request, and a request assignment number for tracking. Replies will be sent by email whenever possible, regardless of how they were received.

Email: **support@gemstone.com**

The preferred method of contact. Please do not send files larger than 100K (for example, core dumps) to this address. A special address for large files will be provided on request.

World Wide Web: **http://www.gemstone.com**

Technical Support is located under Services. A Help Request Form is available for request submissions. This form requires a password, which is free of charge but must be requested by completing the Registration Form, found in the same location. Allow 24 hours for your registration to be recorded and a password assigned.

Facsimile: **(503) 629-8556**

When you send a fax to Technical Support, you should also leave a voicemail message to make sure your fax will be picked up as soon as possible.

We recommend you use telephone contact only for more serious requests that require immediate evaluation, such as a production database that is non-operational.

Telephone: **(800) 243-4772 or (503) 690-3503**

Emergency requests will be handled by the first available engineer. If you are reporting an emergency and you receive a recorded message, do not use the voicemail option. Transfer your call to the operator, who will take a message and immediately contact an engineer.

Non-emergency requests received by telephone will be placed in the normal support queue for evaluation and response.

24x7 Emergency Technical Support

GemStone offers, at an additional charge, 24x7 emergency technical support. This support entitles customers to contact GemStone 24 hours a day, 7 days a week, 365 days a year, if they encounter problems that cause their production application to go down, or that have the potential to bring their production application down. Contact your GemStone account manager for more details.

Contents

Chapter 1. Introduction to GemStone

1.1 Overview of the GemStone System 1-2
1.2 Multi-User Object Server 1-2
1.3 Programmable Server Object System 1-2
1.4 Partitioning of Applications Between Client and Server 1-3
1.5 Large-Scale Repository 1-4
1.6 Queries and Indexes 1-4
1.7 Transactions and Concurrency Control 1-5
1.8 Connections to Outside Data Sources 1-6
1.9 Object Security and Account Management 1-6
1.10 Services to Manage the GemStone Repository 1-7

Chapter 2. Programming With GemStone

2.1 The GemStone Programming Model 2-2
Server-based classes, methods, and objects. 2-2
Client and Server Interfaces 2-2
Gemstone Sessions 2-4

2.2 GemStone Smalltalk . . . 2-5
Language Extensions . . . 2-5
Constraining Variables . . . 2-5
Query Syntax . . . 2-7
Auto-Growing Collections . . . 2-7
Class Library Differences . . . 2-7
No User Interface . . . 2-7
Different File Access . . . 2-7
Different C Callouts . . . 2-8
Class Library Extensions . . . 2-8
More Collection Classes . . . 2-8
RC Classes . . . 2-8
User Account and Security Classes . . . 2-8
System Management Classes. . . . 2-9
File In and File Out . . . 2-9
Inter-Application Communications. . . . 2-9
2.3 Process Architecture . . . 2-10
Gem Process . . . 2-10
Stone Process . . . 2-10
Shared Object Cache . . . 2-10
Scavenger Process. . . . 2-11
Extents and Repositories . . . 2-11
Transaction Log . . . 2-11
NetLDI . . . 2-12
Login Dynamics. . . . 2-12
. . . 2-12

Chapter 3. Name Resolution and Object Sharing

3.1 Sharing Objects . . . 3-2
3.2 UserProfile and Session-based Symbol Lists . . . 3-2
What's In Your Symbol List? . . . 3-3
Examining Your Symbol List . . . 3-4
Inserting and Removing Dictionaries From Your Symbol List . . . 3-6
Updating Symbol Lists . . . 3-8
Finding Out Which Dictionary Names an Object . . . 3-11
3.3 Sharing Objects . . . 3-12
Publishers, Subscribers and the Published Dictionary . . . 3-12

Chapter 4. Collection and Stream Classes

4.1 An Introduction to Collections . . . 4-2
Protocol Common To All Collections . . . 4-4
Creating Instances . . . 4-4
Adding Elements . . . 4-5
Enumerating . . . 4-6
Selecting and Rejecting Elements . . . 4-7
Constraining The Elements Of A Collection . . . 4-8
4.2 Collection Subclasses . . . 4-10
AbstractDictionary . . . 4-10
AbstractDictionary Protocol . . . 4-10
Internal Dictionary Structure . . . 4-10
KeyValueDictionary . . . 4-11
SymbolDictionary . . . 4-12
SequenceableCollection . . . 4-13
Accessing and Updating Protocol . . . 4-14
Adding Objects to SequenceableCollection . . . 4-15
Removing Objects From A SequenceableCollection . . . 4-16
Comparing SequenceableCollection . . . 4-17
Copying SequenceableCollection . . . 4-17
Enumeration and Searching Protocol . . . 4-18
Arrays . . . 4-19
Strings . . . 4-24
Symbols . . . 4-29
DoubleByteString and DoubleByteSymbol . . . 4-30
UnorderedCollection . . . 4-30
Bag . . . 4-30
IdentityBag . . . 4-30
Class IdentitySet . . . 4-37
Set . . . 4-42
4.3 Stream Classes . . . 4-42
Stream Protocol . . . 4-44
Creating Printable Strings with Streams . . . 4-46

Chapter 5. Querying

5.1 Relations . . . 5-2

What You Need To Know 5-3
5.2 Selection Blocks and Selection. 5-4
Selection Block Predicates and Free Variables 5-5
Predicate Terms 5-6
Predicate Operands 5-6
Predicate Operators 5-7
Conjunction of Predicate Terms 5-9
Limits on String Comparisons. 5-10
Redefined Comparison Messages in Selection Blocks 5-10
Changing the Ordering of Instances 5-15
Collections Returned by Selection. 5-16
Streams Returned by Selection 5-16
5.3 Additional Query Protocol 5-19
5.4 Indexing For Faster Access 5-20
Identity Indexes 5-20
Creating Identity Indexes. 5-21
Equality Indexes 5-22
Creating Equality Indexes 5-22
Creating Indexes on Very Large Collections 5-23
Automatic Identity Indexing 5-24
Implicit Indexes 5-24
Indexes and Transactions 5-24
Inquiring About Indexes 5-25
Removing Indexes 5-26
Implicit Index Removal 5-26
Transferring Indexes 5-26
Removing and Re-creating Indexes 5-27
Indexing and Authorization 5-28
Indexing and Performance. 5-28
Indexing Errors 5-29
5.5 Nil Values and Selection 5-30
5.6 Paths Containing Collections 5-31
5.7 Sorting and Indexing 5-34

Chapter 6. Transactions and Concurrency Control

6.1 Gemstone’s Conflict Management 6-2

Transactions 6-2
When Should You Commit a Transaction? 6-2
Reading and Writing in Transactions 6-3
Reading and Writing Outside of Transactions 6-4
6.2 How GemStone Detects Conflict 6-5
Concurrency Management 6-6
Transaction Modes 6-7
Changing Transaction Mode 6-7
Beginning New Manual Transactions 6-8
Committing Transactions. 6-8
Handling Commit Failure In A Transaction 6-10
Indexes and Concurrency Control. 6-10
Aborting Transactions 6-11
Updating the View Without Committing or Aborting 6-12
6.3 Controlling Concurrent Access With Locks 6-13
Locking and Manual Transaction Mode 6-14
Lock Types 6-14
Read Locks 6-14
Write Locks. 6-15
Exclusive Locks 6-15
Acquiring Locks 6-16
Lock Denial 6-17
Dead Locks 6-18
Dirty Locks 6-18
Locking Collections Of Objects Efficiently 6-19
Upgrading Locks 6-21
Locking and Indexed Collections 6-22
Removing or Releasing Locks 6-22
Releasing Locks Upon Aborting or Committing 6-23
Inquiring About Locks 6-24
6.4 Classes That Reduce the Chance of Conflict 6-26
RcCounter 6-27
RcIdentityBag 6-29
RcQueue 6-29
RcKeyValueDictionary 6-31

Chapter 7. Object Security and Authorization

7.1 How GemStone Security Works 7-2
Login Authorization 7-2
The UserProfile 7-3
System Privileges 7-3
Object Level Security 7-3
Segments 7-4
Default Segment and Current Segment 7-6
Objects and Segments 7-7
Read and Write Authorization and Segments 7-9
How GemStone Responds to Unauthorized Access 7-9
Owner Authorization 7-10
Segments in the Repository 7-12
Changing the Segment for an Object 7-13
Revoking Your Own Authorization—a Side Effect 7-16
7.2 An Application Example. 7-16
7.3 A Development Example 7-19
Planning Segments for User Access. 7-21
Protecting the Application Classes 7-21
Planning Authorization for Data Objects 7-21
Planning Groups 7-23
Planning Segments 7-25
Developing the Application 7-25
Setting Up Segments for Joint Development 7-26
Making the Application Accessible for Testing 7-28
Moving the Application into a Production Environment. . . . 7-28
Segment Assignment for User-created Objects 7-29
7.4 Assigning Objects to Segments 7-29
Segments for New Objects 7-29
Removing Segments 7-30
7.5 Privileged Protocol for Class Segment 7-31
7.6 Segment-related Methods 7-32

Chapter 8. Class Versions and Instance Migration

8.1 Versions of Classes 8-2
Defining a New Version 8-3

8.2 ClassHistory 8-3
Defining a Class with a Class History 8-3
Accessing a Class History 8-5
Assigning a Class History 8-6
Class Histories and Constraints 8-6
8.3 Migrating Objects 8-7
Migration Destinations 8-7
Migrating Instances 8-8
Finding Instances and References 8-8
Using the Migration Destination 8-9
Bypassing the Migration Destination 8-10
Migration Errors 8-11
Instance Variable Mappings 8-13
Default Instance Variable Mappings 8-14
Customizing Instance Variable Mappings 8-16
Migrating Collection Class Objects 8-24

Chapter 9. File I/O and Operating System Access

9.1 Accessing Files 9-2
Specifying Files 9-2
Creating a File 9-3
Opening and Closing a File 9-4
Writing to a File 9-5
Reading From a File 9-5
Positioning 9-6
Testing Files 9-7
Removing Files 9-7
Examining A Directory 9-8
9.2 Executing Operating System Commands 9-9
9.3 File In, File Out, and Passive Object 9-10
9.4 Creating and Using Sockets 9-12

Chapter 10. Signals and Notifiers

10.1 Communicating Between Sessions 10-2
10.2 Object Change Notification 10-3

How the Object Server Notifies a Session 10-3
Setting Up a Notify Set 10-5
Adding Objects to a Notify Set. 10-5
Collections 10-7
Listing Your Notify Set 10-8
Removing Objects From Your Notify Set 10-8
Notification of New Objects 10-9
Receiving Object Change Notification 10-10
System | signaledObjects. 10-11
Polling for Changes to Objects. 10-12
Troubleshooting. 10-13
Indexes 10-13
Frequently Changing Objects 10-13
Special Classes 10-13
Methods for Object Notification. 10-15
10.3 Gem-to-Gem Signaling 10-15
Sending a Signal. 10-17
Receiving a Signal 10-19
10.4 Performance Considerations 10-21
Increasing Speed. 10-21
Dealing With Signal Overflow. 10-22
Using Signals and Notifiers with RPC Applications 10-23
Sending Large Amounts of Data. 10-23
Maintaining Signals and Notification When Users Log Out 10-23

Chapter 11. Error Handling

11.1 Signaling Errors to the User 11-1
11.2 Handling Errors in Your Application 11-5
Activation Exceptions 11-6
Static Exceptions 11-6
Defining Exceptions 11-9
Categories and Error Numbers 11-9
Handling Exceptions 11-14
Raising Exceptions 11-15
Flow of Control 11-17
Signaling Other Exception Handlers 11-21
Removing Exception Handlers 11-23

Recursive Errors 11-24
Uncontinuable Errors 11-24

Chapter 12. Tuning Performance

12.1 Clustering Objects for Faster Retrieval 12-2
Will Clustering Solve the Problem? 12-2
Cluster Buckets 12-3
Cluster Buckets and Extents 12-4
Using Existing Cluster Buckets. 12-5
Creating New Cluster Buckets 12-6
Cluster Buckets and Concurrency 12-7
Cluster Buckets and Indexing 12-8
Clustering Objects 12-8
The Basic Clustering Message 12-8
Depth-first Clustering 12-11
Assigning Cluster Buckets 12-11
Clustering vs. Writing to Disk 12-11
Using Several Cluster Buckets 12-11
Clustering Class Objects 12-12
Maintaining Clusters 12-14
Determining an Object's Location 12-14
Why Do Objects Move? 12-15
12.2 Optimizing for Faster Execution 12-15
The Class ProfMonitor 12-15
Profiling Your Code. 12-16
The Profile Report 12-19
Optimization Hints 12-21
12.3 Modifying Cache Sizes for Better Performance 12-24
Configuration File Cache Size Parameters 12-24
Tuning Cache Sizes 12-25
Tuning the Temporary Object Space 12-25
Tuning the Gem Private Page Cache 12-26
Tuning the Stone Private Page Cache 12-26
Tuning the Shared Page Cache. 12-27
12.4 Generating Native Code 12-28
Enabling Native Code 12-28
Limitations of Native Code. 12-29

Chapter 13. Advanced Class Protocol

13.1 Adding and Removing Methods 13-2
Defining Simple Accessing and Updating Methods 13-2
Removing Selectors 13-4
Modifying Classes 13-4
The Basic Compiler Interface 13-5
13.2 Examining a Class's Method Dictionary 13-6
13.3 Examining, Adding, and Removing Categories 13-10
13.4 Accessing Variable Names and Pool Dictionaries 13-13
13.5 Testing a Class's Storage Format 13-16

Appendix A. Basic Smalltalk Syntax

The Smalltalk Class Hierarchy A-1
How to Create a New Class A-2
Case-Sensitivity A-2
Statements A-2
Comments A-3
Expressions A-3
Kinds of Expressions A-4
Literals A-4
Numeric Literals. A-4
Character Literals A-5
String Literals A-6
Symbol Literals A-6
Array Literals A-7
Variables and Variable Names A-7
Declaring Temporary Variables A-8
Pseudovariables A-8
Assignment A-9
Message Expressions A-9
Messages A-10
Reserved Selectors A-10
Optimized Selectors A-10
Messages as Expressions A-11
Combining Message Expressions A-13
Summary of Precedence Rules A-14

Cascaded Messages A-14
Array Constructors A-15
Path Expressions A-17
Returning Values A-18
Blocks. A-19
Blocks with Arguments A-20
Blocks and Conditional Execution. A-22
Conditional Selection A-22
Two-way Conditional Selection A-23
Conditional Repetition A-23
Code Formatting A-25
A.1 Smalltalk BNF A-27

Appendix B. GemStone Error Messages

List of Figures

Figure 3.1. The GsSession Symbol List is a Copy of the UserProfile Symbol List . 3-3
Figure 3.2. Self-Referencing Symbol Dictionary 3-6
Figure 4.1. Simplified Collection Class Hierarchy 4-3
Figure 4.2. SequenceableCollection Class Hierarchy 4-13
Figure 4.3. Employee Relations 4-38
Figure 4.4. Stream Class Hierarchy. 4-43
Figure 5.1. Employee Relation 5-2
Figure 5.2. Anatomy of a Selection Block 5-5
Figure 5.3. Anatomy of a Selection Block Predicate Term. 5-6
Figure 6.1. View States. 6-3
Figure 7.1. User Access to Application Segment1 7-5
Figure 7.2. Multiple Segment Assignments for a Compound Object 7-8
Figure 7.3. User Access to a Segment's Objects 7-10
Figure 7.4. Segments in a GemStone repository. 7-13
Figure 7.5. Application Objects Assigned to Three Segments 7-17
Figure 7.6. User Access to Application Segment1 7-18
Figure 7.7. User Access to Application Segment2 7-19
Figure 7.8. Access Requirements During an Application's Life Cycle 7-20

Figure 7.9. Segments Required for User Access to Application Objects 7-27
Figure 10.1. The Object Server Tracks Object Changes 10-4
Figure 10.2. Communicating from Session to Session 10-16
Figure 11.1. Method Contexts and Associated Exceptions 11-6
Figure 11.2. Defining Error Dictionaries. 11-10
Figure 11.3. Default Flow of Control in Exception Handlers 11-18
Figure 11.4. Activation Exception Handler With Explicit Return 11-20
Figure 11.5. Activation Exception Handler Passing Control to Another Handler . 11-23
Figure 12.1. 12-19
Figure A.1. Smalltalk BNF A-28
Figure A.2. Smalltalk Lexical Tokens A-29

List of Tables

Table 4.1. String's Case-Insensitive Search Protocol 4-25
Table 4.2. String's Case-Sensitive Search Protocol 4-25
Table 5.1. Comparison Operators Allowed in a Selection Block 5-7
Table 6.1. Transaction Conflict Keys 6-9
Table 7.1. Access for Application Objects Required by Users 7-22
Table 7.2. Access to the First Five Objects Through Owner and World Authorization 7-23
Table 7.3. Access to the Last Six Objects Through Owner and World Authorization 7-24
Table 7.4. Access to the Last Six Objects Through the Personnel Group 7-24
Table 7.5. Access to the Last Six Objects Through the Payroll and Sales Groups . 7-25
Table 9.1. GsFile Method Summary 9-4
Table 11.1. GemStone Event Errors. 11-8
Table 11.2. Uncontinuable Errors 11-25
Table 12.1. Clustering Protocol 12-13
Table 12.2. GemNativeCodeMax Values. 12-28
Table 12.3. GemNativeCodeThreshold Values 12-28
Table 13.1. Method Dictionary Access 13-7

Table 13.2. Category Manipulation 13-10
Table 13.3. Access to Variable Names 13-13
Table 13.4. Storage Format Protocol 13-16
Table 0.1. Optimized Selectors A-11

Chapter

1

Introduction to GemStone

This chapter introduces you to the GemStone system. GemStone provides a distributed, server-based, multiuser, transactional Smalltalk runtime system, Smalltalk application partitioning technology, access to relational data, and production-quality scalability and availability. The GemStone object server allows you to bring together object-based applications and existing enterprise and business information in a three-tier, distributed client/server environment.

1.1 Overview of the GemStone System

GemStone provides a wide range of services to help you build objects-based information systems. GemStone:

- is a multi-user object server
- is a programmable server object system
- manages a large-scale repository of objects
- supports partitioning of applications between client and server
- supports queries and indexes for large-scale object processing
- supports transactions and concurrency control in the object repository
- supports connections to outside data sources
- provides object security and account management
- provides services to manage the object repository.

Each of these features is described in greater detail in the following sections.

1.2 Multi-User Object Server

GemStone can support over 1,000 concurrent users, object repositories of up to 100 gigabytes, and sustained object transaction rates of over 100 transactions per second. Server processes manage the system, while user sessions support individual user activities. Repository and server processes can be distributed among multiple machines, and shared memory and SMP can be leveraged.

Multiple user sessions can be active at the same time, and each user may have multiple sessions open. A flexible naming scheme allows separate or shared namespaces for individual users. Coherent groups of objects can be distributed through replication. Changes users make to objects are committed in transactions, with concurrency controls and locks ensuring that multi-user changes to objects are coordinated. Security is provided at several levels, from login authorization to object access privileges.

1.3 Programmable Server Object System

GemStone provides data definition, data manipulation, and query facilities in a single, computationally complete language — GemStone Smalltalk. The GemStone Smalltalk language offers built-in data types (classes), operators, and control structures comparable in scope and power to those provided by languages such as C, C++, or Pascal, in addition

to multi-user concurrency and repository management services. All system-level facilities, such as transaction control, user authorization, and so on, are accessible from GemStone Smalltalk.

This manual discusses the use of GemStone Smalltalk for system and application development, particularly those aspects of GemStone Smalltalk that are unique to running in a multi-user, secure, transactional system. See the *GemStone System Administration Guide* for more information about system administration functions.

1.4 Partitioning of Applications Between Client and Server

GemStone applications can access objects and run their methods from a number of languages, including Smalltalk, C, C++, or any language that makes C calls (such as COBOL or Fortran). Objects created from any of these languages are interoperable with objects created from the other languages, and can run their methods within GemStone.

To provide this functionality, GemStone provides interface libraries of Smalltalk classes, C++ classes and functions, and C functions. These language interfaces, known collectively as GemBuilder, allow you to move objects between an application program and the GemStone repository, and to connect client objects to GemStone objects. GemBuilder also provides remote messaging capabilities, client replicates, and synchronization of changes.

GemBuilder for Smalltalk is a set of classes installed in a client Smalltalk image that provides access to objects in the GemStone repository. The client Smalltalk application can use these classes to gain access to all of GemStone's production capabilities. GemBuilder for Smalltalk also supports *transparent* GemStone access from a Smalltalk application — client Smalltalk and GemStone objects are related to each other, and GemBuilder maintains the relationship and propagates changes between these client Smalltalk and GemStone objects, not the application.

GemBuilder for C is a library of C functions that provide a bridge between an application's C code and the GemStone object repository. You can work with GemStone objects by importing them into the C program using structural access or by sending messages to objects in the repository through GemStone Smalltalk. You can also call C routines from within GemStone Smalltalk methods.

GemBuilder for C++ provides both persistent storage for C++ applications and access to persistent GemStone objects from applications written in C++. Because C++ objects stored in GemStone take on identity and exist independently of the program that created them, they can be used by other applications, including those written in other programming languages.

Your GemStone system includes one or more of these interfaces. Separate manuals available for each of the GemBuilder products provide full documentation of the functionality and use of these products.

1.5 Large-Scale Repository

Object programming languages such as Smalltalk have proven to be highly efficient development tools. Smalltalk exploits inheritance and code reuse and provides the flexibility of modeling real world objects with self-contained software modules. Most Smalltalk implementations, however, are memory based. Objects are either not saved between executions, or they are saved in a primitive manner that does not lend itself to concurrent usage or sharing. Smalltalk programmers save their work in an "image," which is a file that stores their development environment on a workstation. The image holds the application's classes and instances, the compiled code for all executable methods, and the values of the variables defined in the product.

GemStone is based on the Smalltalk object model—like a single-user Smalltalk image, it consists of classes, methods, instances and meta objects. Persistence is established by attaching new objects to other persistent objects. All objects are derived from a named root (AllUsers). Objects that have been attached and committed to the repository are visible to all other users. However, unlike client Smalltalks with memory-based images, the GemStone repository is accessed through disk caches, so it is not limited in size by available memory. A GemStone repository can contain over a billion objects. Repositories can be distributed among many different machines and files. Because each object in a repository has a unique object identifier (known as an OOP—object-oriented pointer), GemStone applications can access any object without having to know its physical location.

1.6 Queries and Indexes

GemStone lets you model information in structures as simple as the data permits, and no more complex than the data demands. You can represent data objects in tables, hierarchies, networks, queues, or any other structure that is appropriate. Each of these objects may also be indexable. Complex data structures can be built by nesting objects of various formats.

The power and flexibility of GemStone Smalltalk allow you to perform regular and associative access queries against very large collections. Because you can represent information in forms that mirror the information's natural structure, the translation of user requests into executable queries can be much easier in GemStone. You do not need to translate users' keystrokes or menu selections into relational algebra formulas, calculus expressions and procedural statements before the query can be executed. See Chapter 5, "Querying.".

1.7 Transactions and Concurrency Control

Each GemStone session defines and maintains a consistent working environment for its application program, presenting the user with a consistent view of the object repository. The user works in an environment in which only his or her changes to objects are visible. These changes are private to the user until the transaction is committed. The effects of updates to the object repository by other users are minimized or invisible during the transaction. GemStone then checks for consistency with other users' changes before committing the transaction.

GemStone provides two approaches to managing concurrent transactions:

- Using the *optimistic* approach, you read and write objects as if you were the only user, letting GemStone manage conflicts with other sessions only when you try to commit a transaction. This approach is easy to implement in an application, but you run the risk of discarding the work you've done if GemStone detects conflicts and does not permit you to commit your transaction. When GemStone looks for conflicts only at your commit time, your chances of being in conflict with other users increase both with the time between your commits and the number of objects being read and written.
- Using the *pessimistic* approach, you prevent conflicts as early as possible by explicitly requesting locks on objects before you modify them. When an object is locked, other users are unable to lock that object or to commit any changes they have made to the object. When you encounter an object that another user has locked, you can wait, or abort your transaction immediately, instead of wasting time doing work that can't be committed. If there is a lot of competition for shared information in your application, or your application can't tolerate even an occasional inability to commit, using locks may be your best choice.

GemStone is designed to prevent conflicts when two users are modifying the same object at the same time. However, some concurrent operations that modify an object, but in consistent ways, should be allowed to proceed. For example, it might not cause any concern if two users concurrently added objects to the same Bag in a particular application.

For such cases, GemStone provides reduced-conflict (Rc) classes that can be used instead of the regular classes in those applications that might otherwise experience too many unnecessary conflicts:

- *RcCounter* can be used instead of a simple number for keeping track of amounts when it isn't crucial that you know the results right away.
- *RcIdentityBag* provides the same functionality as IdentityBag, except that no conflict occurs if a number of users read objects in the bag or add objects to the bag at the same time.

- *RcQueue* provides a first-in, first-out queue in which no conflict occurs when other users read objects in the queue or add objects to the queue at the same time.
- *RcKeyValueDictionary* provides the same functionality as KeyValueDictionary, except that no conflict occurs when users read values in the dictionary or add keys and values to the dictionary at the same time.

See Chapter 6, "Transactions and Concurrency Control."

1.8 Connections to Outside Data Sources

While GemStone methods are all written in Smalltalk (except for a few primitives), you may often want to call out to other logic written in C. GemStone provides a way to attach external code, called userActions, to a GemStone session. With userActions, you can access or generate external information and bring it into GemStone as objects, which can then be committed and made available to other users. GemBuilder for C is used to write userActions in C and add them to GemStone Smalltalk, according to rules described in the *GemBuilder for C* manual. The description of class System in the *GemStone Kernel Reference* describes the messages you can send to invoke these userActions.

GemStone uses this mechanism to build its GemConnect product, which provides access to relational database information from GemStone objects. GemConnect also provides automatic tracking of object modifications for synchronizing the relational database, and supports the generation of SQL to update the relational database with changes. GemConnect is fully encapsulated and maintained in the GemStone object server. Refer to the *GemConnect Programming Guide* for more information about GemConnect and its capabilities.

1.9 Object Security and Account Management

Compared to a single-user Smalltalk system, GemStone requires substantially more security mechanisms and controls. As a tool for server implementation, multi-user Smalltalk must handle requests from many users running a variety of applications, each of which can require different accessibility of objects. Authentication and authorization are the cornerstones of GemStone Smalltalk security.

A server must reliably identify the people attempting to use a system resource. This identification process is known as authentication. Authentication requires a valid user ID and password. Preventing unauthorized users from entering the system by requiring user names and passwords is generally effective against casual intrusion. GemStone Smalltalk supports its own authentication protocol, as well as the Kerberos scheme.

The next type of security, known as authorization, exists within GemStone and controls individual object access. Authorization enforcement is implemented at the lowest level of basic object access to prevent users from circumventing the authorization checking. No object can be accessed from any language without suitable authorization. GemStone provides a number of classes to define and manage object authorization policies. These classes are discussed in greater detail in this manual.

Finally, GemStone defines a set of **privileges** for controlling the use of certain system services. Privileges determine whether the user is allowed to execute certain system functions usually only performed by the system administrator. Privileges are more powerful than authorization. A privileged user can override authorization protection by sending privileged messages to change the authorization scheme.

In GemStone Smalltalk, a user is represented by an instance of class UserProfile. A UserProfile contains the following information about a user:

- unique userID,
- password (encrypted),
- default authorization information,
- privileges,
- group memberships.

Only users who have a UserProfile can log on to the system. For more information on UserProfile, see the *GemStone System Administration Guide*.

See Chapter 7, "Object Security and Authorization."

1.10 Services to Manage the GemStone Repository

GemStone objects are often an enterprise resource. They must be shared among all users and applications to fill their role as repositories of critical business information and logic. Their role goes beyond individual applications, requiring permanence and availability to all parts of the system. GemStone is capable of managing large numbers of objects shared by hundreds of users, running methods that access millions of objects, and handling queries over large collections of objects by using indexes and query optimization. It can support large-scale deployments on multiple machines in a variety of network configurations. All of this functionality requires a wide array of services for management of the repository, the system processes, and user sessions.

GemStone provides services that can:

- support flexible backup and restore procedures,
- recover from hardware and network failures,
- perform object recovery when needed,
- tune the object server to provide high transaction rates by using shared memory and asynchronous I/O processes,
- accommodate the addition of new machines and processors without recoding the system,
- make controlled changes to the definition of the business and application objects in the system.

This manual provides information about programmatical techniques that can be used to optimize your GemStone environment for system administration. Actual system administration and management processes are discussed in the *GemStone System Administration Guide*.

Chapter 2

Programming With GemStone

This chapter provides an overview of the programming environment provided by GemStone.

The GemStone Programming Model
describes how programming in GemStone differs from programming in a client Smalltalk development environment.

GemStone Smalltalk
explains the unique aspects of GemStone Smalltalk that affect programming and application design.

GemStone Architecture
describes GemStone's development and runtime process architecture, and how that architecture influences your programming design and techniques.

2.1 The GemStone Programming Model

GemStone is an object server, so programming with GemStone is somewhat different than programming with a client Smalltalk development environment. However, there is a great deal that GemStone has in common with client Smalltalk development, so many of the programming concepts will be quite familiar to you if you have previously worked with a client Smalltalk system.

Server-based classes, methods, and objects

One key characteristic of GemStone programming is that GemStone Smalltalk runs in a server, not in a client. Running in a server means that GemStone classes and methods are stored in a server-based repository (image), and activated by processes which run on a server, often without a keyboard or screen present. The developer writing GemStone classes and methods is usually working at a client machine, communicating with the GemStone environment remotely.

Running in a server also means that the services provided by GemStone's own class library are oriented toward server activity. GemStone's class library provides functionality for:

- data handling
- collection processing and query processing
- system management
- user account management

The GemStone class library does not provide functionality for screen presentation and user interface issues. User interface functionality is provided in client Smalltalk products.

Because GemStone is an object server, it provides a large number of mechanisms for communicating with GemStone objects from remote machines for development purposes, application support, and system management. Remote machines often host a programming environment that communicates with GemStone through a GemStone interface. A significant part of programming with GemStone is designing the interactions between various client and server-based runtime systems and the GemStone classes, methods, and objects created by the developer.

Client and Server Interfaces

GemStone provides a number of client and server interfaces to make it easy for developers to write applications which make use of GemStone objects, and to write GemStone classes and methods which make use of external data. While an entire application can be built in GemStone Smalltalk and run in the GemStone server, most applications include either a user interface or interaction of some kind with other systems. In addition, management of a running GemStone system involves using GemStone tools and interfaces to program control activities tailored to specific system environments.

GemStone's interfaces are numerous. They include:

GemBuilder for Smalltalk
GemBuilder for Smalltalk consists of two parts: a set of GemStone programming tools, and a programming interface between the client application code and GemStone. GemBuilder for Smalltalk contains a set of classes installed in a client Smalltalk image that provides access to objects in a GemStone repository. Many of the client Smalltalk kernel classes are mapped to equivalent GemStone classes, and additional class mappings can be created by the application developer.

GemBuilder for C++
GemBuilder for C++ provides both shared storage for C++ applications and access to shared objects stored in GemStone by applications written in other languages. GemBuilder for C++ is implemented as a preprocessor based on standard C++ syntax. A class library is provided, giving the programmer a standard set of definitions for commonly used data structures such as sets, arrays, and bags, as well as functions for managing and manipulating GemStone objects with C++ code.

GemBuilder for C
GemBuilder for C is a library of C functions that provide a bridge between an application's C code and the GemStone repository. This interface allows programmers to work with GemStone objects by importing them into the C program using structural access, or by sending messages to objects in the repository through GemStone Smalltalk. C routines can also be called from within GemStone Smalltalk methods.

Topaz
Topaz is a GemStone programming environment that provides keyboard command access to the GemStone object server. Topaz is especially useful for repository administration tasks and batch mode procedures. Because it is command driven and generates ASCII output on standard output channels,

Topaz offers access to GemStone without requiring a window manager or additional language interfaces. You can use Topaz in conjunction with other GemStone development tools such as GemBuilder for C to build comprehensive applications.

UserActions (C callouts from GemStone Smalltalk)
UserActions are similar to user-defined primitives in other Smalltalks. GemBuilder for C can be used to write these user actions, and add them to and execute them from GemStone Smalltalk.

More information about the GemBuilder and Topaz products are found in their respective reference manuals. UserActions are discussed in the *GemBuilder for C* manual.

Gemstone Sessions

All of the GemStone interfaces provide access to GemStone objects and mechanisms for running GemStone methods in the server. This access is accomplished by establishing a session with the GemStone object server. The process for establishing a session is tailored to the language or user of each interface. In all cases, however, this process requires identification of the GemStone object server to be used, the user ID for the login, and other information required for authenticating the login request.

Once a session is established, all GemStone activity is carried out in the context of that session, be it low-level object access and creation, or invocation of GemStone Smalltalk methods.

Sessions allow multiple users to share objects. In fact, different sessions can access the same repository in different ways, depending on the needs of the applications or users they are supporting. For example, an employee may only be able to access employee names, telephone extensions and department names through the human resources application, while a manager may be able to access and change salary information as well.

Sessions also control transactions, which are the only way changes to the repository can be committed. However, a *passive* session can run outside a transaction for better performance and lower overhead. For example, a stock portfolio application that reports the current value of a collection of stocks may run in a session outside a transaction until notified that a price has changed in a stock object. The application would then start a transaction, commit the change, and recalculate the portfolio value. It would then return to a passive session state until the next change notification.

On both UNIX and NT platforms, a session can be integrated with the application into a single process, called a **linked** application. Each session can have only one linked application.

Alternatively, the session can run as a separate process and respond to remote procedure calls (RPCs) from the application. These sessions are called **RPC** applications. PC-based platforms (VisualAge and Visual Smalltalk Enterprise) must run in RPC mode. Sessions may have multiple RPC applications running simultaneously with each other and a linked application.

2.2 GemStone Smalltalk

All Smalltalk languages share common characteristics. GemStone Smalltalk, while providing basic Smalltalk functionality, also provides features that are unique to multi-user, server-based programming.

GemStone Smalltalk provides data definition, data manipulation, and query facilities in a single, computationally complete language. It is tailored to operate in a multi-user environment, providing a model of transactions and concurrency control, and a class library designed for multi-user access to objects. GemStone Smalltalk operates on server-class machines to take advantage of shared memory, asynchronous I/O, and disk partitions. It was built with transaction throughput and client communication as chief considerations.

At the same time, its common characteristics with other Smalltalks allow you to implement shared business objects with the same language you use to build client applications. Since the same code can execute either on the client or on the object server, you can easily move behavior from the client to the server for application partitioning.

GemStone Smalltalk extends standard Smalltalk in several ways.

Language Extensions

Constraining Variables

GemStone Smalltalk allows you to constrain instance variables to hold only certain kinds of objects. The keyword `constraints:` in a class creation message takes an argument that specifies the classes the instance variable will accept. Specifying a constraint is optional.

Constraining a variable ensures that the variable will contain either nil or instances of the specified class or that class's subclasses. When you constrain an instance

variable to be a kind of Array, you guarantee that it will always be an Array, an instance of some subclass of Array (such as InvariantArray), or nil.

Constraining Named Instance Variables

You specify constraints on a class's named instance variables when you create the class. The keyword `constraints:`, a part of the standard subclass creation message, takes an Array of constraints as its argument.

The following example creates a new subclass of Object with three instance variables constrained to be Strings and one to be an Integer.

Example 2.1

```
Object subclass: 'Employee'
  instVarNames: #( 'name' 'job' 'age' 'address')
  inDictionary: UserGlobals
  constraints: #[
            #[#name, String], #[#job, String],
            #[#age, Integer], #[#address, String] ].
```

In this example, `constraints:` takes as its argument an Array of two-element Arrays. The first element is a symbol naming one of the class's instance variables and the second element is a class to which the variable is constrained.

Array constructors (enclosed in brackets) are used here instead of literal arrays (enclosed in parentheses) to build the constraint.

The details of constraint specification differ for named and unordered instance variables. Chapter 4, "Collection and Stream Classes," explains how to constrain unordered instance variables.

Inherited Constraints

Each class inherits instance variables and any constraints on them from its superclass. You can make inherited constraints more restrictive in the subclass by naming the inherited instance variables in the argument to `constraints:` in the creation statement.

The following example creates a subclass of Employee in which the constraint on the instance variable *age* is SmallInteger instead of Integer:

Example 2.2

```
Employee subclass: 'YoungEmployee'
  instVarNames: #()
  classVars: #()
  poolDictionaries: #()
  inDictionary: UserGlobals
  constraints: #[ #[#age, SmallInteger] ]
  isInvariant: false.
```

YoungEmployee's other inherited instance variables, which are not listed in the `constraints:` argument, retain their original constraints.

You can only restrict an inherited instance variable to a subclass of the inherited constraint. So, in the previous example, you could not have constrained `age` to be of class Number or Array, since neither Array nor Number is a subclass of Integer.

Circular Constraints

A circular constraint occurs when an instance variable of a class is constrained to hold instances of its own class, or when each of two classes is constrained to hold instances of the other's class.

Query Syntax

Enterprise applications need to support efficient searching over collections to find all objects that match some specified criteria. Each collection class in GemStone Smalltalk provides methods for iterating over its contents and allowing any kind of complex operation to be performed on each element. All collection classes understand the messages `select:`, `reject:`, and `detect:`.

In GemStone Smalltalk, an index provides a way to traverse backwards along a path of instance variables for every object in the collection for which the index was created. This traversal process is usually much faster than iterating through an entire collection to find the objects that match the selection criteria.

A special query syntax lets you use GemStone Smalltalk's extended mechanism for querying collections with indexes. In addition, the special syntax for select blocks lets you specify a path of named instance variables to traverse during a query.

Auto-Growing Collections

GemStone Smalltalk allows you to create collections of variable length, allowing you to add and delete elements without manually readjusting the collection size. GemStone handles the memory management necessary for this process.

Class Library Differences

No User Interface

GemStone Smalltalk does not provide any classes for screen presentation or user interface development. These aspects of development are handled in your client Smalltalk.

Different File Access

GemStone class GsFile provides a way to create and access non-GemStone files. Many of the methods in GsFile distinguish between files stored on the client machine and files stored on the server machine. GsFile allows the use of full pathnames or environment variables to specify location. If environment variables are used, how the variable is expanded depends on whether the process is running on the client or the server.

Different C Callouts

GemStone Smalltalk uses a mechanism called *user actions* to invoke C functions from within methods. User actions must be written and installed according to special rules, which are described in the *GemBuilder for C* manual.

Class Library Extensions

You can subclass all GemStone-supplied classes, and applications will inherit all their predefined structure and behavior. This manual discusses some of these classes and methods. Your GemBuilder interface provides an excellent means for becoming familiar with the GemStone class hierarchy. A complete description of all GemStone Smalltalk classes is found in the *GemStone Kernel Reference*.

More Collection Classes

GemStone Smalltalk provides a number of specialized Collection classes, such as the KeyValueDictionary classes, that have been optimized to improve application speed and support scaling capability. A full discussion of these classes is found in the Collections chapter of this manual.

RC Classes

Reduced-conflict (RC) classes minimize spurious conflicts that can occur in a multiuser environment. RC classes are used in place of their regular counterpart classes in those applications that you determine may otherwise encounter too many of these conflicts. RC classes do not circumvent normal conflict mechanisms, but they have been specially designed to eliminate or minimize commit errors on operations that analysis has determined are not true conflicts.

User Account and Security Classes

UserProfile is used by GemStone in conjunction with information GemStone gathers during each session to provide a range of security and authorization services, including login authorization, memory and file protection, secondary storage management, location transparency, logical name translation, and coordination of resource use by concurrent users. This manual contains a discussion of how UserProfile is used by GemStone during a session. The *System Administration Guide* contains procedures for creating and maintaining UserProfiles.

Segment is used to control ownership of and access to objects. With Segment, you can abstractly group objects, specify who owns the objects, specify who can read them, and specify who can write them. Each repository is composed of segments. This manual provides a full discussion of segments in the Security chapter.

Both classes are described in detail in the *GemStone Kernel Reference*.

System Management Classes

GemStone Smalltalk provides a number of classes that offer system management functionality. Most of the actions that directly call on the data management kernel can be invoked by sending messages to System, an abstract class that has no instances. All disk space used by GemStone to store data is represented as a single instance of class Repository, and all data management functions, such as extent creation and access, backup and restoration, and garbage collection are performed against this class. The class ProfMonitor allows you to monitor and capture statistics about your application performance that can then be used to optimize and tune your Smalltalk code for maximum performance. The class ClusterBucket can be used to cluster objects across transactions, meaning their receivers will be placed, as far as possible, in contiguous locations on the same disk page or in contiguous locations on several pages.

Implementation of these classes is discussed in this manual. All of these classes are described in detail in the *GemStone Kernel Reference*.

File In and File Out

Smalltalk allows you to file out source code for classes and methods, save the resulting text file, and file it in to another repository. The GemStone class PassiveObject also allows you to file out **objects** and file them in to another repository. This functionality is similar to that provided by VisualWorks' Binary Object Streaming Service (BOSS) and Visual Smalltalk Enterprise's Object Filer. More information about the process is provided in this manual. A description of the PassiveObject class is provided in the *GemStone Kernel Reference*.

Inter-Application Communications

GemStone Smalltalk provides two ways to send information from one currently logged-in session to another:

GemStone can tell an application when an object has changed by sending the application a **notifier** at the time of commit. Notifiers eliminate the need for the application to repeatedly query the Gem for this information. Notification is optional, and can be enabled for only those objects in which you are interested.

Applications can send messages directly to one another by using Gem-to-Gem **signals**. Sending a signal requires a specific action by the receiving Gem.

2.3 Process Architecture

GemStone provides the technology to build and execute applications that are designed to be partitioned for execution over a distributed network. GemStone's architecture provides both scalability and maintainability. Sections describing the main aspects of GemStone architecture follow.

Gem Process

GemStone creates a Gem process for each session. The Gem runs GemStone Smalltalk and processes messages from the client session. It provides the user with a consistent view of the repository, and it manages the user's GemStone session, keeping track of the objects the users has accessed, paging objects in and out of memory as needed, and performing dynamic garbage collection of temporary objects. The Gem performs the bulk of commit processing. A user application is always connected to at least one Gem, and may have connections to many Gem. Gems can be distributed on multiple, heterogeneous servers, which provides distribution of processing and SMP support. The Gem also offers users the ability to link in user primitives for customization.

Stone Process

The Stone process is the resource coordinator. One Stone process manages one repository. It synchronizes activities and ensures consistency as it processes requests to commit transactions. Individual Gem processes communicate with the Stone through interprocess channels. The Stone:

- coordinates commit processing,
- coordinates lock acquisition,
- allocates object IDs,
- allocates object Pages,
- writes transaction logs.

Shared Object Cache

The shared object cache provides efficient retrieval of objects from disk, and the ability for multiple Gems to access the same object. When modified, an object is written to a new location in the cache. Memory is managed and allocated on a page basis. The cache also contains buffers for communications between Gems and the Stone. The shared cache monitor initializes the shared memory cache, manages cache allocation to the sessions, and dynamically adjusts this allocation to fit the workload. It also makes sure that frequently accessed objects remain in memory, and that large objects queries do not flush data from the cache. These controls allow complex applications to be run on the same repository by multiple users with no degradation in performance.

Scavenger Process

The scavenger process dynamically reclaims space used by unreferenced objects. This process is sometimes called dynamic garbage collection, and in GemStone, may be referred to as the GC Gem. The scavenger process also dynamically defragments the repository while maintaining requested object clustering. It has a multi-level collection architecture, consisting of:

- Dynamic cleanup of temporary objects,
- Epoch cleanup of shared objects, and
- Full sweep of the repository.

Extents and Repositories

Extents are composed of multiple disk files or raw partitions. A repository, which is the logical storage unit in which GemStone stores objects, is actually an ordered file of one or more extents. Extents can be distributed to heterogeneous servers. Objects can be clustered on an extent for efficient storage and access.

Extents can be mirrored for improved fault tolerance. By mirroring extents, you store each object in two places to reduce the chance of data loss. GemStone automatically stores each newly committed object in both locations. Any damage to one extent leaves all the objects intact in the mirrored extent, allowing GemStone to automatically switch over to the active mirrored extent on an extent fault. Using mirrored extents can also improve distributed query performance. GemStone allows the creation of one mirrored extent for each extent in the repository.

Transaction Log

GemStone's transaction log provides complete point-in-time roll-forward recovery. The tranlog contents are composed by the Gem, and the Stone writes the tranlog using asynchronous I/O. Commit performance is improved through I/O reduction, since only log records need to be written, not many object pages. In addition, the object pages stay in memory to be reused. Log files may also be mirrored for fault tolerance. GemStone supports both file based and raw device configuration of tranlogs.

NetLDI

In a distributed system, each machine that runs a Stone monitor, Gem session process, or linked application, or on which an extent resides, must have its own network server process, known as a NetLDI (Network Long Distance Information). A NetLDI reports the location of GemStone services on its machine to remote processes that must connect to those services. The NetLDI also spawns other GemStone processes on request.

Login Dynamics

When you log in to GemStone, GemStone establishes for you a logical entity called a GsSession, which is comparable to an operating system session, job, or process. GemStone creates a separate instance of GsSession each time a user logs in, and it monitors, serves, and protects each session independently.

You can log into GemStone through any of its interfaces. Whichever interface you use, GemStone requires the presentation of a user ID (a name or some other

identifying string) and a password. If the user ID and password pair match the user ID and password pair of someone authorized to use the system, GemStone permits interaction to proceed; if not, GemStone severs the logical connection.

The system administrator (or a user with equivalent privileges) assigns each GemStone user an instance of class UserProfile, which contains, among other information, the user ID and password. GemStone uses the UserProfile to establish logical names and default locations, resolve references to system objects, and perform similar tasks. The system administrator gives each new UserProfile appropriate customized rights, and stores it with a set of all other UserProfiles in the set AllUsers.

You can obtain your own UserProfile by sending a message to System. Class UserProfile defines protocol for obtaining information about default names, privileges, and so forth. More information about UserProfile is provided in this manual. Class UserProfile is described in the *GemStone Kernel Reference*, while procedures for creating and maintaining UserProfile are found in the *GemStone System Administration Guide*.

The GemStone system administrator can also configure a GemStone system to monitor failures to log in, to note repeated login attempts, and to disable a user's account after a number of failed attempts to log into the system through that account. The *GemStone System Administration Guide* describes these procedures in greater detail.

Chapter 3

Name Resolution and Object Sharing

This chapter describes how Smalltalk finds the objects to which your programs refer and explains how you can arrange to share (or not to share) objects with other GemStone users.

Sharing Objects
explains how Smalltalk allows users to share objects of any kind.

The Session-based and UserProfile Symbol Lists
describes the mechanism that the Smalltalk compiler uses to find objects referred to in your programs.

Specifying Who Can Share Which Objects
discusses how you can enable other users of your application to share information.

3.1 Sharing Objects

Smalltalk permits concurrent access by many users to the same data objects. For example, all Smalltalk programmers can make references to the kernel class Object. These references point directly to the single class Object—not to copies of Object.

GemStone allows shared access to objects without regard for whether those objects are files, scalar variables, or collections representing entire databases. This ability to share data facilitates the development of multi-user applications.

To find the object referred to by a variable, GemStone follows a well-defined search path:

1. The local variable definitions: temporary variables and arguments
2. Those variables defined by the class of the current method definition: instance, class, class instance, or pool variables
3. The symbol list assigned your current session

If GemStone cannot find a match for a name in one of these areas, you are given an error message.

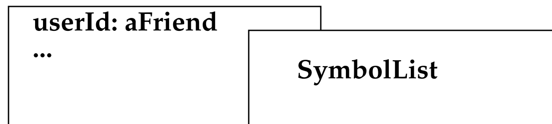
3.2 UserProfile and Session-based Symbol Lists

The GemStone system administrator assigns each GemStone user an object of class UserProfile. Your UserProfile stores such information as your name, your encrypted password, native language, and access privileges. Your UserProfile also contains the instance variable *symbolList*.

When you log in to GemStone, the system creates your current session (which is an instance of GsSession object) and initializes it with a copy of the UserProfile SymbolList object. Smalltalk refers to this copy of the symbol list to find objects you name in your application.

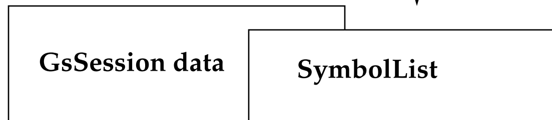
Figure 3.1 The GsSession Symbol List is a Copy of the UserProfile Symbol List

Persistent UserProfile:



At Log in, GsSession creates a copy of the symbolList in Your UserProfile

Transient data:



This instance of GsSession is not copied into any client interface nor committed as a persistent object. Since the symbolList is transient, changes to it cannot incur concurrency conflicts. Changes to the current session's symbolList do not affect the UserProfile symbolList, allowing the UserProfile symbolList to continue to serve as a default list for other logins. At the same time, methods are provided to synchronize your session and UserProfile symbolLists.

What's In Your Symbol List?

The data curator adds to your UserProfile symbol list the SymbolDictionaries containing associations defining the names of all the objects he or she thinks you might need. Although the decision about which objects to include is entirely up to the data curator, your symbol list contains at least:

- A "system globals" dictionary called *Globals*. This dictionary contains some or all of the Smalltalk kernel classes (Object, Class, Collection, etc.) and any other objects to which all of your GemStone users need to refer. Although you can read the objects in *Globals*, you are probably not permitted to modify them.
- A private dictionary in which you can store objects for your own use and new classes you do not need to share with other GemStone users. That private dictionary is usually named *UserGlobals*.

The symbol list may also include special-purpose dictionaries that are shared with other users, so that you can all read and modify the objects they contain. The data

curator can arrange for a dictionary to be shared by inserting a reference to that dictionary in each user's UserProfile symbol list.

Except for the dictionaries Globals and UserGlobals, the contents of each user's SymbolList are likely to be different.

Examining Your Symbol List

To get a list of the dictionaries in your persistent symbol list, send your UserProfile the message `dictionaryNames`. For example:

Example 3.1

```
System myUserProfile dictionaryNames
 1 UserGlobals
 2 UserClasses
 3 ClassesForTesting
 4 Globals
 5 Published
```

The SymbolDictionaries listed in the example have the following function:

- UserGlobals

  Contains application and application service objects. This dictionary is the default used by GemBuilder for Smalltalk to replicate classes in GemStone.

- UserClasses

  Contains per-user class definitions, and is created by the GemBuilder for Smalltalk to replicate classes when necessary. Putting this dictionary before the Globals dictionary allows an application or user to override Kernel classes without changing them. Keeping it separate from UserGlobals allows a distinction between classes and application objects.

- ClassesForTesting

  A user-defined dictionary.

- Globals

  Provides access for the GemStone Kernel Classes.

- Published

  Provides space for globally-visible shared objects created by a user.

To list the contents of a symbol list Dictionary:

- If you are using Topaz, set your display level to 2, and execute some expression that returns the Dictionary. “Display level” settings are available only in Topaz.
- If you are running GemBuilder, select the expression UserGlobals in a GemStone workspace and execute `GS-inspect`.

Example 3.2

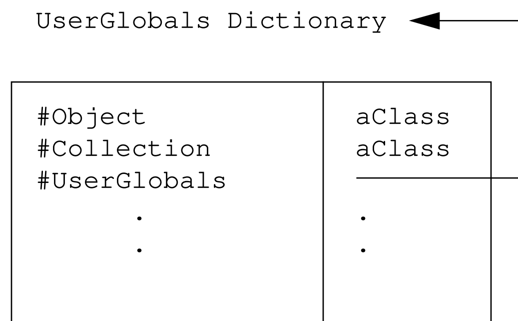
```
UserGlobals
 self
      #NativeLanguage
      #UserGlobals
      #GcUser
```

If you examine all of your symbol list dictionaries, you’ll see that most of the kernel classes are listed. In addition, you may notice objects called CompileError, RuntimeError, FatalError, AbortingError, WeekDayNames, and MonthNames. These objects provide the text for error messages, days of the week, and months in your native language.

Finally, you’ll discover that most of the dictionaries refer to themselves. Since the symbol list must contain all source code symbols that are not defined locally nor by the class of a method, the symbol list dictionaries need to define names for themselves so that you can refer to them in your code. Figure 3.2 illustrates that the dictionary named UserGlobals contains an association for which the key is UserGlobals and the value is the dictionary itself.

The object server searches symbol lists sequentially, taking the first definition of a symbol it encounters. Therefore, if a name, say “#BillOfMaterials,” is defined in the first dictionary and in the last, Smalltalk finds only the first definition.

Figure 3.2 Self-Referencing Symbol Dictionary



Inserting and Removing Dictionaries From Your Symbol List

Creating a dictionary is like creating any other object, as the following example shows. Once you've created the new dictionary, you can add it to your symbol list by sending your UserProfile the message `insertDictionary:` *aSymbolDict* `at:` *anInt*.

Example 3.3

```
| newDict |
  newDict := SymbolDictionary new.
  newDict at: #NewDict put: newDict.
  System myUserProfile insertDictionary: newDict at: 1.
```

As you might expect, `insertDictionary: at:` shifts existing symbol list dictionaries as needed to accommodate the new dictionary. In this example, the new dictionary is inserted into the UserProfile symbolList and then updated in the current session.

Because the Smalltalk compiler searches symbol lists sequentially, taking the first definition of a symbol it encounters, your choice of the index at which to insert a new dictionary is significant.

The following example places the object myCollection in the user's private dictionary named `myClassDict`. Then it inserts myClassDict in the first position of the current Session's symbolList, which causes the object server to search

myClassDict prior to UserGlobals, meaning the GemStone object server will always find `myCollection` in myClassDict.

Example 3.4

```
| myClassDict |
(System myUserProfile resolveSymbol:#myClassDict) isNil
  ifTrue:[
     myClassDict := (System myUserProfile createDictionary:
                       #myClassDict).
  ]
  ifFalse:[
    myClassDict := (System myUserProfile resolveSymbol:
                       #myClassDict) value
  ].
Object subclass: 'myCollection'
       instVarNames: #('this' 'that' 'theOther')
       classVars: #()
       poolDictionaries: #()
       inDictionary: myClassDict
       constraints: #()
       isInvariant: false
%

GsSession currentSession userProfile insertDictionary: myClassDict
at: 1.
%

"Create a new object named myCollection,
 placed in the UserGlobals dictionary: "

Object subclass: 'myCollection'
       instVarNames: #('snakes' 'snails' 'tails')
       classVars: #()
       poolDictionaries: #()
       inDictionary: UserGlobals
       constraints: #()
       isInvariant: false
%
```

When you refer to myCollection, only the version of the object that is in myClassDict is returned, because the object server returns the first occurrence found when searching the dictionaries listed by the current session's symbol list. If the UserGlobals dictionary is listed before myClassDict, the object server only finds the version of myCollection in UserGlobals.

You may redefine any object by creating a new object of the same name and placing it in a dictionary that is searched before the dictionary in which the matching object resides. Therefore, inserting, reordering or deleting a dictionary from the symbol list may cause the GemStone object server to return a different object than you may expect.

This situation also happens when you create a class with a name identical to one of the kernel class names.

CAUTION
We strongly recommend that you do not redefine any kernel classes, as their implementation may change from one version of GemStone to the next. Creating a subclass of a kernel class to redefine or extend that functionality is usually more efficient.

To remove a symbol list dictionary, send your UserProfile the message `removeDictionaryAt:` *anInteger*. For example:

Example 3.5

```
System myUserProfile removeDictionaryAt: 1
```

Updating Symbol Lists

There are many ways that the current Session's symbol list can get out of sync with the UserProfile symbol list. As some of the examples in this chapter show, updates can made to the current session symbolList that exist only as long as you are logged in. By changing only the symbol list for the current session, you can dynamically change the session namespace without causing concurrency conflict.For example, if you are developing a new class, you may purposely set your current session symbol list to include new objects for testing.

Three UserProfile methods help synchronize the persistent and transient symbol lists:

- UserProfile | insertDictionary *aDictionary* at: *anIndex*

  This method inserts a Dictionary into the UserProfile symbol list at the specified index.

- UserProfile | removeDictionary: *aDictionary*

  This method removes the specified dictionary from the UserProfile symbolList.

- UserProfile | symbolList: *aSymbolList*

  This method replaces the UserProfile symbol list with the specified symbol list; However, you cannot execute this method unless you are authorized to write in certain protected portions of the system.

When any of these methods are used, the method modifies the UserProfile symbol list. Then, if the receiver is identical to “GsSession currentSession userProfile”, the current session’s symbolList is updated. If a problem occurs during one of these methods, the persistent symbol list is updated, but the transient current session symbol list is left in its old state.

The following example provides an instruction for copying the transient symbol list into the persistent UserProfile symbol list. The example continues with adding a new dictionary to the current session and finally resets the current session's symbol list back to the UserProfile symbol list.

Example 3.6

```
! Copies the GsSession symbol list to the UserProfile
System myUserProfile symbolList:
                   (GsSession currentSession symbolList copy)
! Checks that the symbol lists are the same
GsSession currentSession symbolList =
                              System myUserProfile symbolList
%
! Adds a new dictionary to the currentsession
GsSession currentSession symbolList add: SymbolDictionary new
%
! compares the two symbollists; they should differ
GsSession currentSession symbolList =
                              System myUserProfile symbolList
%
! Updates the UserProfile symbolList to current session
GsSession currentSession symbolList replaceElementsFrom:
(System myUserProfile symbolList)
%
```

Finding Out Which Dictionary Names an Object

To find out which dictionary defines a particular object name, send your UserProfile the message `symbolResolutionOf:` *aSymbol*. If *aSymbol* is in your symbol list, the result is a string giving the symbol list position of the dictionary defining *aSymbol*, the name of that dictionary, and a description of the association for which *aSymbol* is a key. For example:

Example 3.7

```
"Which symbol list Dictionary defines the object 'Bag'?"
System myUserProfile symbolResolutionOf: #Bag
3 Globals
    Bag  Bag
```

If *aSymbol* is defined in more than one dictionary, `symbolResolutionOf:` finds only the first reference. Smalltalk considers two symbols with the same name to be identical.

To find out which dictionary stores a name for an object and what that name is, send your UserProfile the message `dictionaryAndSymbolOf:` *anObject*. This message returns an array containing the first dictionary in which *anObject* is stored, and the symbol which names the object in that dictionary.

Example 3.8 uses `dictionaryAndSymbolOf:` to find out which dictionary in the symbol list stores a reference to class DateTime:

Example 3.8

```
| anArray theDict myUserPro n |
myUserPro := System myUserProfile. "get the UserProfile"
"Find the Dictionary containing DateTime"
anArray := myUserPro dictionaryAndSymbolOf: DateTime.
theDict := anArray at: 1.

aSymbolDictionary (#DateTime->DateTime,...)
```

Note that `dictionaryAndSymbolOf:` returns the *first* dictionary in which *anObject* is a value.

3.3 Sharing Objects

As you know, all GemStone users have access to such objects as the kernel classes Integer and Collection because those objects are referred to by a dictionary (usually called Globals) that is present in every user's symbol list.

If you want GemStone users to share other objects as well, you need to arrange for references to those objects to be added to the users' symbol lists. You can add the references to the Published Dictionary, which is a GemStone provided dictionary and which has a corresponding segment (PublishedSegment), or you can create another dictionary and authorization segment using the PublishedSegment as a model. The Published Dictionary and Published Segment is not currently used by GemStone classes, but may be utilized by future products.

Your system's authorization mechanism is probably set up to preclude you from modifying UserProfiles other than your own, so you probably need the cooperation of your GemStone system administrator to place a Dictionary in a set of UserProfiles.

Publishers, Subscribers and the Published Dictionary

The Published Dictionary, PublishedSegment, and the groups Subscribers and Publishers together provide an example of how to set up a system for sharing objects.

The Published Dictionary is an initially empty dictionary referred to by your UserProfile. It contains symbols that most users need to access. The objects named in this dictionary are created in the PublishedSegment. The PublishedSegment is owned by the Data Curator and has World access set to none. Two groups have access to the Published Segment:

- Subscribers have read-only access to the PublishedSegment, and
- Publishers have read-write access to the PublishedSegment.

Publishers can create objects in the PublishedSegment and enter them in the Published Dictionary. Then members of the Subscribers group can access the objects.

Chapter 4

Collection and Stream Classes

The Collection classes make up the largest group of classes in Smalltalk. This chapter describes the common functionality available for Collection classes.

An Introduction to Collections
introduces the Smalltalk objects that store groups of other objects.

Collection Subclasses
describes several kinds of ready-made data structures that are central to Smalltalk data description and manipulation.

Stream Classes
describes classes that add functionality to access or modify data stored as a Collection.

4.1 An Introduction to Collections

Collections can store groups of other objects in indexed or unnamed instance variables. In addition, most classes in the Collection hierarchy can also have named instance variables. Collections can be classified by the orders in which they store elements, the kinds of objects they can store, and the kinds of access methods they provide. A simplified structure of the Collection class hierarchy is listed in Figure 4.1.

How you wish to access information determines which subclasses you choose to create for your objects:

- Access by Key — the Dictionary Classes

  Keys can be numbers, strings, symbols, or any objects that respond meaningfully to the comparison message =. A dictionary is a collection of values which can be accessed by their associations.

  Dictionaries can have named instance variables, if you choose to define them.

- Access by Position — the SequenceableCollection Classes

  You can refer to the component objects of a SequenceableCollection with numeric keys, just as you refer to array elements in C or Pascal by means of numeric subscripts. This Class includes Arrays, Strings, and the Sorted Collection.

  ByteArray, CharacterCollection, and CharacterCollection subclasses store byte values only, while the other sequenceable collections can have named instance variables if you choose to define them.

- Access by Value — the UnorderedCollection Classes

  The objects in these collections are accessed by matching an unnamed instance variable value. These Classes act as black boxes; they hide the internal ordering of their elements from you and from other objects. Bags and Sets are included in the UnorderedCollection Class.

  You may create index structures for fast access to the contents of these classes.

Figure 4.1 Simplified Collection Class Hierarchy

```
Collection
   AbstractDictionary
         Dictionary
         KeyValueDictionary
               IdentityKeyValueDictionary
                     GsMethodDictionary
                     IdentityDictionary
                           SymbolDictionary
                     SymbolKeyValueDictionary
               IntegerKeyValueDictionary
               StringKeyValueDictionary
   SequenceableCollection
         Array
               AbstractCollisionBucket
                     CollisionBucket
                           IdentityCollisionBucket
                           RcCollisionBucket
               InvariantArray
               Repository
               SymbolList
         ByteArray
         CharacterCollection
               DoubleByteString
                     DoubleByteSymbol
               String
                     InvariantString
                     Symbol
         Interval
         OrderedCollection
               SortedCollection
   UnorderedCollection
         Bag
         IdentityBag
               IdentitySet
                     ClassSet
                     StringPairSet
                     SymbolSet
         Set
```

Protocol Common To All Collections

The superclass of the collection classes, Collection, provides some protocol shared by all collection subclasses. In fact, providing that common protocol is Collection's only function; it is an abstract superclass. Instances of Collection itself are not typically useful.

Collection defines methods that enable you to:

- Create instances of its subclasses
- Add and remove elements in collections
- Convert from one kind of class to another
- Enumerate (loop through), compare, and sort the content of collections
- Select or reject certain elements on the collection based on specified criteria.

The GemBuilder interface provides an excellent means for reviewing the purpose and format for each of the categories of methods available for manipulating Collection Classes and subclasses. The examples that follow provide a starting point for using Collections.

All the protocol displayed in the examples is defined in the *GemStone Kernel Reference* manual.

Creating Instances

All Collection classes respond to the familiar instance creation message `new`. When sent to a Collection class, this message causes a new instance of the class with no elements (size zero) to be created. Most kinds of collections can expand as you add additional objects.

Another instance creation message, `new:` *anInteger*, causes any Collection subclass except IdentityBag or IdentitySet to create an instance with *anInteger* nil elements:

Example 4.1

```
| myArray |
myArray := Array new: 5.
myArray at: 3 put: 'a string'.
myArray size
5
```

It's sometimes slightly more efficient to use `new:` than `new`, because a Collection created with `new:` need not expand repeatedly as you add new elements.

Class Collection defines an additional instance creation message, `withAll:`*aCollection*, that creates a new instance of the receiver containing all of the objects stored in *aCollection*. For example:

Example 4.2

```
| birds |
birds := Array withAll:#('wren' 'robin' 'turkey buzzard').
birds at: 3
turkey buzzard
```

Adding Elements

Collection defines for its subclasses two basic methods for adding elements:

- the `add:` method adds one element to the Collection
- the `addAll:` method adds several elements to the Collection at once.

The following example uses both these methods to add elements to an instance of Collection's subclass IdentitySet. (An IdentitySet is an unordered, extensible collection of objects—you'll learn about its properties in detail later.)

Example 4.3

```
| potpourri |
potpourri := IdentitySet new.
UserGlobals at: #Potpourri put: potpourri.

Potpourri add: 'a string of characters'; add: 0.0035;add: #aSymbol.
Potpourri addAll: #(#flotsam #jetsam #salvage).
Potpourri
%
```

IdentitySet is a very simple kind of collection, so adding elements is straightforward. Other Collection classes override these methods in order to control access to elements or to enforce an ordering scheme. Still other subclasses of Collection provide additional methods that add elements at numbered positions or symbolic keys. You'll read about those specialized methods later.

Enumerating

Collection defines several methods that enable you to loop through a collection's elements. Because iterating or enumerating the elements of a data structure is one of the most common programming tasks, Collection's built-in enumeration facilities are extremely useful; they relieve you of worrying about data structure size and loop indexes. And because they have been carefully tailored to each of Collection's specialized subclasses, you needn't create a custom iterative control structure for each enumeration problem.

The most general enumeration message is `do:` *aBlock*. When you send a Collection this message, the receiver evaluates the block repeatedly, using each of its elements in turn as the block's argument.

Suppose that you made an instance of IdentitySet in this way:

Example 4.4

```
| virtues |
virtues := IdentitySet new.
virtues addAll: #('humility' 'generosity' 'veracity'
                 'continence' 'patience').
((UserGlobals at: #Virtues put: virtues) sortAscending: '')
   verifyElementsIn: #[ 'continence', 'generosity',
      'humility', 'patience', 'veracity' ]
```

To create a single String to which each virtue has been appended, you could use the message `do:` *aBlock* like this:

Example 4.5

```
| aString |
aString := String new. "Make a new, empty String."
"Append a virtue, followed by a space, to the new String"

(Virtues sortAscending: '')
    do: [:aVirtue | aString := aString , ' ' , aVirtue].
^ aString
 continence generosity humility patience veracity
```

In this example, the method for `do:` executes the body of the block (`aString , ' ' , aVirtue`) repeatedly, substituting each of Virtues' elements in turn for the

block argument *aVirtue*, until all of the virtues have been appended to aString. (The String concatenation message (`","`) is explained later in this chapter.)

In addition to `do:` *aBlock*, Collection provides several specialized enumeration methods.When sent to SequenceableCollections, those messages that return collections (such as `select:`) always preserve the ordering of the receiver in the result. That is, if element *a* comes before element *b* in the receiver, then element *a* is guaranteed to come before *b* in the result.

NOTE:
To avoid unpredictable consequences, do not add elements to or remove them from a collection during enumeration.

Selecting and Rejecting Elements

The messages `select:` *aBlock* and `reject:` *aBlock* make it easy to pick out those elements of a collection that meet some condition and to store them in a new collection of the same kind as the original.

The following examples form two new sets, one containing the virtues 'patience' and 'continence', the other containing all of the other virtues.

Example 4.6

```
| a b |
"Select all of the virtues equal to 'patience' or 'continence'"
a := Virtues select: [:n | (n = 'patience') | (n = 'continence')].
a.

an IdentitySet
 #1 patience
 #2 continence
| a b |
"Select all of the virtues NOT equal to 'patience' or 'continence'"
b := Virtues reject: [:n | (n = 'patience') | (n = 'continence')].
b.
%
an IdentitySet
 #1 humility
 #2 generosity
 #3 veracity
```

Constraining The Elements Of A Collection

The unordered or indexed elements of a collection cannot be constrained individually. They are instead constrained to contain only a single kind of element.

You can specify the kind of elements a collection can hold when you create its class. If you do not, your subclass inherits only the most general kind of constraint—every element of a collection must be a kind of Object.

The following example creates a subclass of IdentitySet whose instances must contain only instances of a version of Employee or a subclass thereof:

Example 4.7

```
Object subclass: 'Employee'
       instVarNames: #( 'name' 'jobclass' 'address')
       classVars: #()
       poolDictionaries: #()
       inDictionary: UserGlobals
       constraints: #[ #[#name,String],
                       #[#jobclass,Integer],
                       #[#address,String] ]
       isInvariant: false

IdentitySet subclass: 'SetOfEmployees'
  instVarNames: #()
  classVars: #()
  poolDictionaries: #()
  inDictionary: UserGlobals
  constraints: Employee
  isInvariant: false.
%
```

When you create a constrained Collection class that has no named instance variables, the `constraints:` keyword takes as its argument a single class name. Because all elements (unnamed instance variables) in a constrained Collection must be of a single kind (or nil), no more information is needed. However, a SetOfEmployees could store a subclass of Employees as well as Employees. A SetOfEmployees could also contain instances of a previous or future version of Employee or its subclass, as long as the class of those instances shares a class history with either class Employee or its subclass.

If the class whose variables you are constraining also defines named instance variables, the argument to the `constraints:` keyword is an Array of two-element Arrays followed by a single class name. As with non-Collection classes, the first element of each two-element Array is a Symbol naming one of the class's instance variables, and the last element of each is a class. The final class name supplies the constraint for the unnamed instance variables.

For example, suppose you want define a SetOfEmployees as an unordered collection of employees, each with a named instance variable representing a different division of your company. In that case, the instance creation message would look like Example 4.8.

Example 4.8

```
IdentitySet subclass: 'SetOfEmployees'
 instVarNames: #( 'division')
  classVars: #()
  poolDictionaries: #()
  inDictionary: UserGlobals
  constraints: #[ #[#division, String], Employee ]
  isInvariant: false
```

Inquiring about Constraints

The `allConstraints` method returns all of the constraints on a class. It returns an array of classes, with one element in the array for each named instance variable in the class. If the receiver has constraints on the instance variables, the array returned by the method appends those constraints as additional elements.

The following example shows how the returned array looks for SetOfEmployees defined in Example 4.8. The first four elements are the four instance variables for SetOfEmployees: name, jobclass, address, and division. Then the constraint for division is listed as an element. The last element in the returned array is the constraint for SetOfEmployees to contain only instances of Employee.

Example 4.9

```
SetOfEmployees allConstraints verifyElementsIn:
   #[ Object, Object, Object, Object, String, Employee ]
true
```

To determine the constraint on indexed or unordered instance variables of a class, send it the `varyingConstraint` message:

Example 4.10

```
SetOfEmployees varyingConstraint
Employee
```

Also useful are the methods (inherited from Behavior) `constraintOn:`, `definition`, and `hierarchy`.

4.2 Collection Subclasses

This chapter describes the properties of Collection's concrete subclasses, and it gives you some guidance about choosing places for new classes that you might want to add to the Collection hierarchy.

Subclasses of Collection can be grouped by the kinds of access methods they provide and the kinds of objects their instances can store. Let's first consider those collection classes that don't provide access to elements through external numeric indexes.

AbstractDictionary

AbstractDictionary is a subclass of Collection. AbstractDictionary requires that all of an instance's elements must have unique keys.

The subclasses of AbstractDictionary provide access to their elements by means of keys that can be strings, symbols, integers, or objects of any kind.

AbstractDictionary Protocol

AbstractDictionary defines a large number of methods that enable you to store and retrieve objects on the basis of either keys or values. Some of the methods return only single keys or values, while others return entire associations.

Internal Dictionary Structure

Dictionaries provide their special facilities by storing key-value pairs instead of simple, linear lists of objects. Many of the messages that dictionaries understand

are specialized for referring to either the key or the value portions of their component associations.

In the following example, the message `includesKey:` *aKey* tests to see whether the dictionary myDictionary contains the definition of *glede*:

Example 4.11

```
run
| myDictionary |
myDictionary := StringKeyValueDictionary new.
myDictionary at: 'glede' put: 'a bird of prey'.
(myDictionary includesKey: 'glede') ifTrue:
              [ myDictionary at: 'glede' ].
%
a bird of prey
```

KeyValueDictionary

KeyValueDictionary has several subclasses, divided according to the type of key used to access the information:

- IdentityKeyValueDictionary,
- IntegerKeyValueDictionary; and
- StringKeyValueDictionary.

In each case, the hashing function is applied to the key.

SymbolDictionary

A subclass of IdentityKeyValueDictionary, SymbolDictionary, constrains all of its keys to be symbols, which it stores in instances of class SymbolAssociation.

The following example creates a new instance of SymbolDictionary called “Lizards,” then stores some strings at symbolic keys.

Example 4.12

```
| Lizards |
Lizards := SymbolDictionary new.
Lizards at: #skink put: 'a small, berry-eating lizard'.
Lizards at: #gecko put: 'a harmless, nocturnal lizard'.
Lizards at: #komodo put: 'a big, irascible reptile'.
Lizards at: #monitor put: 'a large reptile that lives in
your roommate''s closet and usually doesn''t bite'.

"Access one of the SymbolDictionary elements:"
Lizards at: #skink
 a small, berry-eating lizard
```

The `at:put:` message in this example took a symbol as its first argument instead of (as with sequenceable collections) an integer.

To retrieve a value from a dictionary, you need only send it the message `at:` *aKey*. At the end of the previous example, `#skink` is a key.

It’s important to understand that, just as the entry for “2” is not necessarily the second item in the dictionary on your bookshelf, the numeral 2 does not signify anything about position when used as a key in a Smalltalk dictionary. Like strings, symbols, and other dictionary keys, numerals identify but do not locate dictionary values.

This simple protocol for storing and retrieving objects on the basis of symbolic instead of positional keys finds wide use in Smalltalk. In fact, the Smalltalk compiler and interpreter take advantage of dictionaries to resolve symbols, store methods, and retrieve error messages, as well as other tasks.

SequenceableCollection

Unlike the AbstractDictionary collections, the SequenceableCollections let you refer to their elements with integer indexes, and they understand messages such as `first` and `last` that refer to the order of those indexed elements. The SequenceableCollection classes differ from one another mainly in their literal representations, the kinds of elements they store, and the kinds of changes they permit you to make to their instances.

Figure 4.2 is an abbreviated diagram of the SequenceableCollection family tree. It depicts the SequenceableCollection classes you are likely to use as general-purpose data structures.

Figure 4.2 SequenceableCollection Class Hierarchy

```
SequenceableCollection
      Array
      ByteArray
      CharacterCollection
            DoubleByteString
                  DoubleByteSymbol
            String
                  Symbol
      Interval
      OrderedCollection
            SortedCollection
```

SequenceableCollection is an abstract superclass. The methods it establishes for its concrete subclasses let you read, write, copy, and enumerate collections in ways that depend on ordering.

For example, there are methods that enable you to read or write an element at a particular index, to ask for an element's index, to request the first and last elements of a collection, and to copy specified parts of one collection to another.

Accessing and Updating Protocol

Class Object defines the messages `at:` *anIndex* and `at:` *anIndex* `put:` *anObject*. The class SequenceableCollection interprets these messages as referring to elements whose positions are identified by integer keys.

The following example uses `at:` and `at:put:` to read and write elements of an Array.

Example 4.13

```
| colors |
colors := Array new.
colors at: 1 put: 'vermilion'.
colors at: 2 put: 'scarlet'.
colors at: 3 put: 'crimson'.
colors at: 2
scarlet
```

Most of the time, SequenceableCollection can grow to accommodate new objects. However, you must store each new item at an index no more than one greater than the largest index you've already used. In the previous example, this requirement permits you to add a color at index 4, but not at index 7. The subsection, "Creating Arrays" on page 4-20, explains a feature for creating large arrays with nil elements. Initializing the array with nil values enables you to store new objects wherever you want. The following example uses other methods defined by SequenceableCollection:

Example 4.14

```
| anArray |
anArray := Array new.
anArray at: 1 put: 'string one';
        at: 2 put: 'string two';
        at: 3 put: 'string three'.
anArray first.
string one
anArray last.
string three
anArray indexOf: (anArray at: 2)
 2
```

Adding Objects to SequenceableCollection

SequenceableCollection defines two `new` methods for adding objects to its instances.

The message `addLast:` *anObject* appends its argument to the receiver, increasing the size of the receiver by one. For example, given the array *anArray*:

Example 4.15

```
anArray addLast: 'string four'.
anArray size.
4
anArray last.
string four
```

The message `insert:` *aSequenceableCollection* `at:` *anIndex* inserts the elements of a new SequenceableCollection into the receiver at *anIndex* and returns the receiver. For example:

Example 4.16

```
| colors moreColors |
colors := Array new add: 'red'; add: 'blue';
                    add: 'green'; yourself.
moreColors := Array new add: 'mauve'; add: 'taupe';yourself.
colors insert: moreColors at: 2.
colors verifyElementsIn:
                    #('red' 'mauve' 'taupe' 'blue' 'green')
 an Array
 #1 red
 #2 mauve
 #3 taupe
 #4 blue
 #5 green
```

If *anIndex* is exactly one greater than the size of the receiver, this method appends each of *aSequenceableCollection's* elements to the receiver.

In addition to the two new adding methods, SequenceableCollection redefines `add:` so it puts objects only at the end of the receiver. In other words, `add:` does the same thing as `addLast:`.

Removing Objects From A SequenceableCollection

You can remove a one or more objects from a SequenceableCollection. In the following example, `deleteObjectAt:` removes the first element of the array `rockClingers`, decreasing the array's size by one:

Example 4.17

```
| rockClingers |
rockClingers := Array withAll: #('limpet' 'mussel' 'whelk').
UserGlobals at: #rockClingers put: rockClingers.
(rockClingers deleteObjectAt: 1) = 'limpet'
  ifFalse:[ ^ 'wrong deletion result'
  ].

rockClingers verifyElementsIn: #( 'mussel' 'whelk')
 an Array
 #1 mussel
 #2 whelk
```

The next example removes the rest of `rockClinger`'s elements, leaving an array of size zero:

Example 4.18

```
rockClingers deleteFrom: 1 to: 2.
rockClingers verifyElementsIn: #()
an Array
```

Comparing SequenceableCollection

SequenceableCollection redefines the comparison methods inherited from Object so that those methods take into account the classes of the collections to be compared and the number and order of their elements. Here are the conditions that must be met for two SequenceableCollections to be considered equal:

- The classes of the two SequenceableCollections must be the same.
- The two SequenceableCollections must be of the same size.
- Corresponding elements of the two objects must be equal.

You can, of course, create subclasses of SequenceableCollections in which you implement comparison messages with different behavior.

Copying SequenceableCollection

SequenceableCollection understands two copying messages—one that returns a sequence of the receiver's elements as a new collection, and one that copies a sequence of the receiver's elements into an existing SequenceableCollection.

The following example copies the first two elements of an InvariantArray to a new InvariantArray:

Example 4.19

```
| tropicalMammals |
tropicalMammals:= #('capybara' 'tapir' 'margay')
      copyFrom: 1 to: 2.
tropicalMammals verifyElementsIn: #('capybara' 'tapir')
an InvariantArray
#1 capybara
#2 tapir
```

The next example copies two elements of an array into a different array, overwriting the target array's original contents:

Example 4.20

```
| numericArray |
numericArray := Array new add: 1; add: 2;
                          add: 99; add: 88; yourself.
#( 1 2 3 4 ) copyFrom: 3 to: 4 into: numericArray startingAt: 3.
numericArray verifyElementsIn: #( 1 2 3 4 )
an Array
 #1 1
 #2 2
 #3 3
 #4 4
```

Bear in mind that copies of SequenceableCollection, like most Smalltalk copies, are "shallow." In other words, the elements of the copy are not simply equal to the elements of the receiver—they are the same objects.

Enumeration and Searching Protocol

Class SequenceableCollection redefines the enumeration and searching messages inherited from Collection in order to guarantee that they process elements in order, starting with the element at index 1 and finishing with the element at the last index.

SequenceableCollection also defines a new enumeration message, `reverseDo:`, which acts like `do:` except that it processes the receiver's elements in the opposite order.

SequenceableCollections understand `findFirst:` *aBlock* and `findLast:` *aBlock*. The message `findFirst:` returns the index of the first element that makes *aBlock* true, while `findLast:` returns the index of the last. For example, given tropicalMammals as defined in the last example:

Example 4.21

```
tropicalMammals findFirst: [:aMammal | aMammal = 'capybara']
1
```

Arrays

As you have seen in previous examples, instances of Array and of its subclasses contain elements that you can address with integer keys that describe the positions of Array elements. For example, `myArray at: 1` refers to the first element of myArray. Example 4.22 uses Array indexing, with protocol from Number, Block, and Boolean, to code a classic sorting algorithm for a subclass of Array:

Example 4.22

```
method: SubArray
sortAscending
| selfSize tempStorage exchangeMade |
exchangeMade := true.
selfSize := (self size) - 1.
[ exchangeMade ] whileTrue:
   [exchangeMade := false.
      1 to: selfSize do: [ :n |
          ((self at: n ) > (self at: n + 1))
             ifTrue: [tempStorage := self at: n.
                      self at: n put: (self at: 1 + n).
                      self at: n+1 put: tempStorage.
                      exchangeMade := true.  ].  ].  ].
^self
%
run  "See that the bubble sort works"
(SubArray withAll: #( 9 7 5 3 1 2 4 6 8 ))
   sortAscending verifyElementsIn: #( 1 2 3 4 5 6 7 8 9 )
%
true
```

One of the most important differences between Smalltalk arrays and a GemStone array is that GemStone arrays are extensible; you can add new elements to an array at any time. However, it is usually most efficient to create arrays that are initially large enough to hold all of the objects you may want to add.

Creating Arrays

You are free to create an array with the inherited message `new` and let the array lengthen automatically as you add elements. However, arrays created with `new` initially allocate very little storage. As you add objects to such an array, it must lengthen itself to accommodate the new objects.

Therefore, you will often want to create your arrays with the message `new:` *aSize* (inherited from class Behavior), which makes a new instance of the specified size:

```
| tenElementArray |
tenElementArray := Array new: 10.
```

The selector `new:` stores nil in the indexed instance variables of the empty array. Having created an array with enough storage for the elements you intend to add, you can proceed to fill it quickly.

Changing the Size of an Existing Array

As you've seen, a SequenceableCollection can grow or shrink automatically at run time as you add or delete elements. However, it's also possible for you to change the size without explicitly storing or removing elements, using the message `size:` inherited from class Object.

In the following example, `size:` increases the length of an array to 500 and then decreases it to zero.

Example 4.23

```
| anArray |
anArray := Array new.
anArray size: 500.
anArray size: 0
```

When you lengthen an array with `size:`, the new elements are set to nil.

Example 4.24 uses `size:` in a simple implementation of a Stack class:

Example 4.24

```
Array subclass: 'Stack'
      instVarNames: #()
      inDictionary: UserGlobals
category: 'Stack Management'
method: Stack
push: anObject
self add: anObject
%
method: Stack
pop
| theTop |
theTop := self last.
self size: (self size - 1).
^theTop
%
method: Stack
clear
self size: 0
%
method: Stack
top
^self last
%
run
"See that it works"
#[ Stack new push: #one; push: #two; push: #three; pop;
push: #four; pop; pop ]
verifyElementsIn: #( #two )
%
```

Efficient Implementations of Large Arrays

When you create an array of slightly over 2000 elements with `new:`, or when you add enough new elements to grow an array to this size using `size:`, the new elements are not set to nil, as that would waste storage. Instead, GemStone uses a sparse tree implementation to make more efficient use of resources. This behavior occurs in a manner that is transparent to you, and you can place values into the new elements of the array in the same manner as you would with smaller arrays.

Representing Arrays Literally—Invariant Arrays

Earlier, you encountered literal arrays that looked like this:

```
#('element one' 'element two' 'element three')
```

Although this example was referred to as a literal array, the compiler actually translates such entities into an object of class InvariantArray.

Constraints in the Indexed Part of an Array

The following example creates a subclass of Array constrained to hold numbers:

Example 4.25

```
Array subclass: 'Stack'
  instVarNames: #() classVars:#() poolDictionaries:#[]
  inDictionary: UserGlobals
  constraints:#[ Number ]
  instancesInvariant: false
  isModifiable: false
```

The following example creates a class with constraints on both named and indexed instance variables. The instance variable ''name'' is constrained to hold a string; the indexed portion is constrained to hold numbers.

Example 4.26

```
Array subclass: 'NamedStack'
  instVarNames: #( 'name')
  classVars:#()
  poolDictionaries:#[]
  inDictionary: UserGlobals
  constraints:#[ #[ 'name', String ], Number ]
  instancesInvariant: false
  isModifiable: false
```

The following example shows an equivalent way to create NamedStack. This example creates a modifiable class, modifies it to add constraints, and then changes the class to a normal class.

Example 4.27

```
Array subclass: 'NamedStack'
  instVarNames: #( 'name' )
  inDictionary: UserGlobals
  isModifiable: true )
instVar: 'name' constrainTo: String ;
varyingConstraint: Number ;immediateInvariant
```

Strings

A String is a SequenceableCollection modified to accept only instances of Character as elements. Class String expands the protocol it inherits from SequenceableCollection to include messages specialized for comparing, searching, concatenating, and changing the case of character sequences.

Class String and its subclasses are all *byte objects*. A byte object has two important practical implications:

- You cannot create a String subclass that has named instance variables.
- When you use `new:` to create an instance of a kind of String, Smalltalk sets the new instance's indexed instance variables to null (ASCII 0).

Creating Strings

You have already seen many strings created as literals. In addition to creating strings literally, you can use the instance creation methods inherited from String's superclasses:

Example 4.28

```
| myString |
myString := String withAll: #($a $z $u $r $e).
myString
azure
```

Many of String's other inherited messages are also useful:

Example 4.29

```
'azure' last   "return the String's last character"
$e

'azure'indexOf:$z  "return the position of $z in the String"
2
```

Searching and Pattern Matching Strings

Class String defines methods that can tell you whether a string contains a particular sequence of characters and, if so, where the sequence begins. The Class String contains methods for case-sensitive and case-insensitive search and

compare. Table 4.1 describes those messages for case-insensitive strings, Table 4.2 describes those messages for case-sensitive strings.

Table 4.1 String's Case-Insensitive Search Protocol

Message	Description
at: *anIndex* equalsNoCase: *aCharCollection*	Returns true if *aCharCollection* is contained in the receiver, starting at *anIndex*. Returns false otherwise.
findPattern: *aPattern* startingAt: *anIndex*	Searches the receiver, beginning at *anIndex*, for a substring that matches *aPattern*. If a matching substring is found, returns the index of the first character of the substring. Returns zero (0) otherwise. The argument *aPattern* is an Array containing zero or more Strings plus zero or more occurrences of the special wild card characters $* or $?. The character $? matches any single character in the receiver, and $* matches any sequence of characters in the receiver.

Table 4.2 String's Case-Sensitive Search Protocol

Message	Description
at: *anIndex* equals: *aCharCollection*	Returns true if *aCharCollection* is contained in the receiver starting at *anIndex*. Returns false otherwise. Generates an error if *aCharCollection* is not a kind of CharacterCollection, or if *anIndex* is not a SmallInteger.
match: *aPrefix*	Returns true if the argument, *aPrefix*, is a prefix of the receiver. Returns false otherwise. The value for *aPrefix* may include the wild card characters $* or $?. The character $? matches any single character in the receiver, and $* matches any sequence of characters in the receiver.
includes: *character*	Returns true if the receiver contains *character*.
indexOf: *aCharacter* startingAt: *startIndex*	Returns the index of the first occurrence of *aCharacter* in the receiver, not preceding *startIndex*. Returns zero (0) if no match is found.

Here is an example using a wild card:

Example 4.30

```
'weimaraner' matchPattern: #('w' $* 'r')
 true
```

This example returns true because the character $* is interpreted as “any sequence of characters.” Similarly, the following example returns the index at which a sequence of characters beginning and ending with $r occurs in the receiver.

Example 4.31

```
'weimaraner' findPattern: #('r' $* 'r') startingAt: 1
6
```

If either of the wild card characters occurs in the receiver, it is interpreted literally. The following expression returns false because the character $* in the receiver is interpreted literally:

Example 4.32

```
"Wildcard characters are literal"
'w*r' matchPattern: #('weimaraner')
 false
```

Comparing Strings

The Class String has methods for comparing strings, divided into categories of Case-Insensitive Comparisons and Case-Sensitive Comparisons. The following example shows the boolean value returned for each comparison.

As shown in the examples, the comparisons for '=' and for `match:` are always case sensitive, while the other messages are case insensitive.

Example 4.33

```
'A' = 'a'                "Case Sensitive compare"
false
'A' match: 'a'           "Case Sensitive compare"
false
'A' < 'a'
true
'A' > 'a'
false
'A' <= 'a'               "Case Insensitive compare"
true
'A' >= 'a'               "Case Insensitive compare"
false
'A' isEquivalent: 'a'    "Case Insensitive compare"
true
'A' equalsNoCase: 'a'    "Case Insensitive compare"
true
'A' < 'b'                "Case Insensitive compare"
true
'b' > 'A'                "Case Insensitive compare"
true
```

Concatenating Strings

A string responds to the message `#,` *someCharacters* by returning a new string in which *someCharacters* (a string, a character, or an array of characters) have been appended to the string's original contents. For example:

Example 4.34

```
'String ' , 'con' , 'catenation'
 String concatenation
```

Although this technique is handy when you need to build a small string, it's not very efficient. In the last example, Smalltalk creates a String object for the first literal, `'String'`. The #, message returns a new instance of String containing

'String con', which is in turn passed to the #, message again to create a third string.

When you need to build a longer string, you'll find it more efficient to use `addAll:` or `add:` (they're the same for class String) like this:

Example 4.35

```
| resultString |
resultString := String new.
resultString add: 'String ';
             add: 'con';
             add: 'catenation'.
resultString

String concatenation
```

Efficient Implementations of Large Strings

When you create a string of over 8000 characters, characters without values are not set to ASCII null, as that would waste storage. Instead, GemStone uses a sparse tree implementation to make more efficient use of resources. This behavior occurs in a manner that is transparent to you, and you can put new characters in the string in the same manner as you would with smaller strings.

Converting Strings To Other Kinds of Objects

Class String defines messages that let you convert a string to an upper- or lowercase string, to a symbol, or to a floating-point number.

Example 4.36

```
'ABCDE' asLowercase
abcde
'abcde' asUppercase
ABCDE
'abcde' asSymbol
abcde'15' asFloat = 1.5e1
true'15' asFloat = 1.5E1
true
```

Literal and nonliteral InvariantStrings and Strings behave differently in identity comparisons. Each nonliteral String (created, for example, with `new`, `withAll:`, or `asString`) has a unique identity. That is, two Strings that are equal are not necessarily identical. For example:

Example 4.37

```
| nonlitString1 nonlitString2 |
nonlitString1 := String withAll: #($a $b $c).
nonlitString2 := String withAll: #($a $b $c).
(nonlitString1 == nonlitString2)
false
```

However, literal strings (InvariantStrings created literally) that contain the same character sequences and are compiled at the same time are both equal and identical:

Example 4.38

```
  | litString1 litString2 |
litString1 := 'abc'.
litString2 := 'abc'.
(litString1 == litString2)
true
```

This distinction can become significant in building sets. Because a set does not accept more than one element with the same identity, you cannot add both *litString1* and *litString2* to the same set. You can, however, store both *nonlitString1* and *nonlitString2* in a single set.

Symbols

Class Symbol is a subclass of String. Smalltalk uses symbols internally to represent variable names and selectors. All symbols are stored in the DataCurator segment, and they may be viewed by all users. All private information should be maintained in Strings, not in Symbols.

You create a symbol using the `withAll:` method. Once a symbol is created, it may not be modified. When you use the `withAll:` method to create a new symbol, Smalltalk checks to see whether the symbol exists in its view of AllSymbols. If the

symbol already exists, the OOP for that symbol is returned, otherwise a new OOP is returned.

DoubleByteString and DoubleByteSymbol

The DoubleByteString and DoubleByteSymbol classes provide the functionality of String and Symbol classes for DoubleByte character sets.

UnorderedCollection

The class UnorderedCollection implements protocol for indexing, which in turn allows for large collections to be queried and sorted efficiently.

All subclasses of UnorderedCollection do not allow nil elements. The repository will silently ignore attempts to create nil elements in these classes.

Chapter 5, “Querying,” describes the querying/sorting functions in detail. The following section describes the protocol implemented in UnorderedCollection’s subclasses.

Bag

A Bag is the simplest unordered collections, made of an aggregation of unordered instance variables. Bags, like most other collections, are elastic, growing to accommodate new objects as you add them.

You access a Bag’s elements by equality. That is, if a variable has the same value as an element that is in the Bag, that element is equal to the variable. If you have two elements in the Bag with the same value, the first element encountered is always returned.

If the Bag contains elements that are themselves complex objects, determining the equality is complex and therefore slower than you might have hoped.

You may constrain the variables in a Bag. The equality-accessed class Bag is provided for compatibility with client Smalltalk standards. If you anticipate a large number of elements in a Bag, we recommend you use the class IdentityBag.

IdentityBag

IdentityBag has faster access and can more easily handle constrained variables. Like a Bag, an IdentityBag is elastic and can hold objects of any kind. An IdentityBag can hold up to 2^{30}-1 (about a billion) objects.

To access an IdentityBag, you rely on the identity (OOP) of the object. This is a much less time-consuming task than an equality comparison, and in most cases it should be sufficient for your design.

If two elements in the IdentityBag have the same OOP, the first element encountered is always returned. The object returned is not guaranteed to be the same one over time.

Because IdentityBag is not ordered, class IdentityBag disallows the inherited message `at:put:`. The inherited messages `add:` and `addAll:` work pretty much as they do with other kinds of collection, except, of course, that they are not guaranteed to insert objects at any particular positions.

IdentityBag defines one new adding message, `add:` *anObject* `withOccurrences:` *anInteger*. This message enables you to add several identical objects to an IdentityBag with a single message:

Example 4.39

```
| aBag |

aBag := IdentityBag new add: 'chipmunk' withOccurrences: 3.
aBag occurrencesOf: 'chipmunk'
3
```

Accessing an IdentityBag's Elements

Since an IdentityBag's elements are not ordered, IdentityBag must disallow the message `at:`. Usually, you'll need to use Collection's enumeration protocol to get at a particular element of a IdentityBag.

The following example uses `detect:` to find a IdentityBag element equal to 'agouti':

Example 4.40

```
| bagOfRodents myRodent |
 bagOfRodents := IdentityBag withAll: #('beaver' 'rat' 'agouti').
 myRodent := bagOfRodents detect: [:aRodent | aRodent = 'agouti'].
 myRodent
 agouti
```

Removing Objects from a IdentityBag

Class IdentityBag provides several messages for removing objects from an identity collection. The message `remove:ifAbsent:` lets you execute some code of your choice if the specified object cannot be found, in this example the message returns false if it cannot find "2" in the IdentityBag:

Example 4.41

```
| myBag |
myBag := IdentityBag withAll: #(2 3 4 5).
((myBag remove: 2 ifAbsent: [^false]) sortAscending: '')
   verifyElementsIn: #[3,4,5]
true
```

Similarly, `removeAllPresent:` *aCollection* is safer than `removeAll:` *aCollection*, because the former method does not halt your program if some members of *aCollection* are absent from the receiver.

All the removal messages act to delete specific objects from an IdentityBag by identity; they do not delete objects that are merely equal to the objects given as arguments. The previous example works correctly because the SmallInteger 2 has a unique identity throughout the system. By way of contrast, consider the following example:

Example 4.42

```
| myBag array1 array2 |
"Create two objects that are equal but not identical,
 and add one of them to a new IdentityBag."
array1 := Array new add: 'stuff'; add:'nonsense' ; yourself.
array2 := Array new add: 'stuff'; add:'nonsense' ; yourself.
"Create an IdentityBag containing array1."
myBag := IdentityBag new add: array1.
UserGlobals at: #MyBag put: myBag.
"Now try to remove one of the objects from the IdentityBag
by referring to its equal twin in the argument to
remove:ifAbsent"
myBag remove: array2 ifAbsent: ['Sorry, can''t find it'].
 Sorry, can't find it
```

Comparing IdentityBags

Class IdentityBag redefines the selector = in such a way that it returns true only if the receiver and the argument:

- are of the same class,
- have the same number of elements,
- have the same constraints on their elements,
- contain only identical (==) elements, and
- contain the same number of occurrences of each object.

Union, Intersection, and Difference

Class IdentityBag provides three messages that perform functions reminiscent of the familiar set union, set intersection, and set difference operators. There is one significant difference between these messages and the set operators — IdentityBag's messages consider that either the receiver or the argument can contain duplicate elements. The description of class IdentityBag in the *GemStone Kernel Reference* provides more information about how these messages behave when the receiver's class is not the same as the class of the argument.

Sorting IdentityBag

Class IdentityBag defines methods that can sort collection elements with maximum efficiency. Sort keys are specified as paths, and they are restricted to paths that are able to bear equality indexes. (See Appendix A, "Basic Smalltalk Syntax." Section: "Path Expressions" on page A-17, for a description of paths. See "Equality Indexes" on page 5-22, for a description of equality indexes.)

The following code defines an Employee object, and an subclass of IdentityBag for containing Employees. Following examples add instances of Employee to the IdentityBag and then sorts those instances:

Example 4.43

```
(Object subclass: 'Employee'
       instVarNames: #( 'name' 'job' 'age' 'bday' 'address')
       classVars: #()
       poolDictionaries: #()
       inDictionary: UserGlobals
       constraints: #[ #[#name, String],  #[#job, String],
                    #[#age,SmallInteger],#[#address,String],
                    #[#bday, DateTime] ]
       isInvariant: false) name
%
Employee compileAccessingMethodsFor:
 #('name' 'job' 'age' 'bday' 'address')
(Employee subclass: 'SymbolNameEmployee'
         instVarNames: #()
         classVars: #()
         poolDictionaries: #()
         inDictionary: UserGlobals
         constraints: #[ #[#name, Symbol] ]
         isInvariant: false) name
%
(IdentityBag subclass: 'BagOfEmployees'
    instVarNames: #()
    classVars: #()
    poolDictionaries: #()
    inDictionary: UserGlobals
    constraints: Employee
    isInvariant: false) name
%
```

The following code creates a few instances of Employee, places them in an IdentityBag subclass named `BagOfEmployees`, and sorts by `sortDescending:`

Example 4.44

```
"Make some Employees,and store them in a BagOfEmployees."
| Conan Lurleen Fred myEmployees |
Conan := (SymbolNameEmployee new) name: #Conan;
   job: 'librarian'; age: 40; address: '999 W. West'.
Fred := (SymbolNameEmployee new) name: #Fred;
   job: 'clerk'; age: 40; address: '221 S. Main'.
Lurleen := (SymbolNameEmployee new) name: #Lurleen;
   job: 'busdriver'; age: 24; address: '540 E. Sixth'.
myEmployees := BagOfEmployees new.
myEmployees add: Fred; add: Lurleen; add: Conan.
UserGlobals at: #myEmployees put: myEmployees
| result |
result := Array new.
(myEmployees sortDescending: 'name') do:
             [:e | result add: e.name ].
result verifyElementsIn: #( #Lurleen #Fred #Conan )
myEmployees sortDescending: 'name'.

an Array
#1 an Employee
  name            Lurleen
#2 an Employee
  name            Fred
#3 an Employee
  name            Conan
```

The messages `sortAscending:` and `sortDescending:` return arrays of elements sorted by a specified instance variable of the element class.

In sorting instances of Float, NaN is regarded as greater than an ordinary floating-point number.

To sort a bag constrained to contain only simple values (such as strings, symbols, numbers, instances of DateTime, or characters), give an empty path as the argument to `sortAscending:` or `sortDescending:`

Example 4.45

```
| myBagOfStrings |
myBagOfStrings := IdentityBag new
                  add: 'alpha'; add: 'beta'; yourself.
(myBagOfStrings sortAscending: '')
   verifyElementsIn: #( 'alpha' 'beta' )

an Array
 #1 alpha
 #2 beta
```

Either of IdentityBag's sorting methods can take an array of paths as its argument. The first path in the array is taken as the primary sort key and the others are taken in order as subordinate keys, as shown in Example 4.46:

Example 4.46

```
| returnArray tempString |
tempString := String new.
returnArray := myEmployees sortAscending: #('age' 'name').
"Build a printable list of the sorted ages and names"
returnArray do: [:i | tempString add: (i age asString);
                                 add: ' '; add: i name;
                                 add: Character lf].
tempString
%
 24 Lurleen
 40 Conan
 40 Fred
```

Here Employees are ordered initially by 'age', the primary sort key. The two Employees who have the same age are ordered by 'name', the secondary sort key.

You may sort a collection on as many as keys as you need. However, the more keys you sort on, the longer the sort will take (in general).

To sort in ascending order on some keys while sorting in descending order on others, use `sortWith:` *anArray*. The argument to this message is an Array of paths alternating with *sort specifications*.

Example 4.47 uses `sortWith:` to sort on age in ascending order and on name in descending order:

Example 4.47

```
| returnArray tempString |
tempString := String new.
returnArray := myEmployees sortWith: #('age' 'Ascending'
                                'name' 'Descending').
returnArray do: [:i | tempString add: (i age asString);
                                add: ' '; add: i name;
                                add: Character lf].
tempString
%
 24 Lurleen
 40 Fred
 40 Conan
```

Class IdentitySet

IdentitySet is similar to IdentityBag, except that IdentitySet does not accept duplicate (that is, identical) elements. You may find sets useful for modeling such entities as relations, which must contain only unique tuples.

To access an IdentitySet, you rely on the identity (OOP) of the object. This is a much less time-consuming task than an equality comparison, and in most cases it should be sufficient for your design.

Because IdentitySet is not ordered, class IdentitySet disallows the inherited messages `at:` and `at:put:`. The inherited messages `add:` and `addAll:` work pretty much as they do with other kinds of Collection, except, of course, that they are not guaranteed to insert objects at any particular positions.

IdentitySet As Relations

Suppose that you wanted to build and query a relation such as the one shown in Figure 4.3:

Figure 4.3 Employee Relations

Employees			
Name	Job	Age	Address
Fred	clerk	40	221 S. Main
Lurleen	busdriver	24	540 E. Sixth
Conan	librarian	40	999 W. West

In Smalltalk, it would be natural to represent such a relation as an IdentitySet of objects of class Employee, with each Employee containing instance variables *name*, *job*, *age*, and *address*. Each element of the IdentitySet corresponds to a tuple, and each instance variable of an element corresponds to a field.

To make it easy to retrieve values from a tuple, you can define methods for class Employee so that an Employee returns the value of its *name* instance variable upon receiving the message `name`, the value of its *age* variable upon receiving the message `age`, and so on.

The examples on the following pages create a small employee relation as described above and show how you might use Collection's enumeration protocol to formulate queries about the relation.

Example 4.48

```
Object subclass: 'Employee'
  instVarNames:
         #('name' 'job' 'age' 'address' 'lengthOfService' )
  classVars: #( )
  poolDictionaries: #[ ]
  inDictionary: UserGlobals
  constraints: #( )
  isInvariant: false.
%
IdentitySet subclass: 'SetOfEmployees'
  instVarNames: #( )
  classVars: #( )
  poolDictionaries: #[ ]
  inDictionary: UserGlobals
  constraints: Employee
  isInvariant: false.
%
! ----- Create Some Instance Methods for Employee -----
category: 'Accessing'
method: Employee
name                    "returns the receiver's name"
    ^name
%
method: Employee
job                     "returns the receiver's job"
    ^job
%
method: Employee
age                     "returns the receiver's age"
    ^age
%
method: Employee
address                 "returns the receiver's address"
   ^address
%
```

```
! -------  More Methods for Employee ---------
category: 'Updating'
method: Employee
name: aNameString      "sets the receiver's name"
      name := aNameString
%
method: Employee
job: aJobString        "sets the receiver's job"
      job := aJobString
%
method: Employee
age: anIntegerAge      "sets the receiver's age"
      age := anIntegerAge
%
method: Employee
address: aString       "sets the receiver's address"
      address := aString
%
category: 'Formatting'
method: Employee
asString
"Returns a String with info about the receiver (an
Employee)."
 ^ (self name) , ' ' ,(self job) , ' ' ,
   (self age asString), ' ' , (self address)
%
method: SetOfEmployees
asTable
"Prints a set of Employees, one to a line"
| aString |
aString := String new.
self do: [:anEmp |
  aString addAll: anEmp asString; add: Character lf .
  ].
^aString
%
expectvalue Employee
run
Employee compileAccessingMethodsFor: Employee.instVarNames
%
```

The following code creates some instances of class Employee and stores them in a new instance of class SetOfEmployees:

Example 4.49

```
"Make some Employees, and store them in a SetOfEmployees."
| Conan Lurleen Fred myEmployees |
Conan := (Employee new) name: 'Conan'; job: 'librarian';
                        age: 40; address: '999 W. West'.
Fred := (Employee new)  name: 'Fred'; job: 'clerk';
                        age: 40; address: '221 S. Main'.
Lurleen := (Employee new) name: 'Lurleen'; job: 'busdriver';
                        age: 24; address: '540 E. Sixth'.
myEmployees := SetOfEmployees new.
myEmployees add: Fred; add: Lurleen; add: Conan.
"Store the Employees in your userglobals dictionary."
UserGlobals at: #myEmployees put: myEmployees.
```

Now it’s possible to form some queries using Collection’s enumeration protocol:

Example 4.50

```
| age40Employees |
"Use select: to ask for employees aged 40."
age40Employees := myEmployees select:
                  [:anEmp | anEmp age = 40].
age40Employees asTable
%
Conan librarian 40 999 W. West
Fred clerk 40 221 S. Main

| conanEmps |
"Ask for employees named 'Conan'"
conanEmps := myEmployees select:
             [:anEmp | anEmp name ='Conan'].
conanEmps asTable
%
Conan librarian 40 999 W. West
```

Example 4.51

```
! More examples of queries for the Collection protocol

| notConanBut40Emps |
"Get employees who are 40 years old and not named Conan."
notConanBut40Emps := myEmployees select:
  [:anEmp | (anEmp age = 40) & (anEmp name ~= 'Conan')].
notConanBut40Emps asTable
%
Fred clerk 40 221 S. Main
%
| youngerThan40Emps |
"Find the employees who are younger than 40."
youngerThan40Emps := myEmployees select:
                         [:anEmp | (anEmp age)< 40].
youngerThan40Emps asTable
%
Lurleen busdriver 24 540 E. Sixth
```

Set

A Set is another unordered collection. Like the Class Bag, an element of a Set is accessed by equality. Unlike a Bag, a Set cannot have multiple objects of the same value.

A Set may have constrained instance variables, but at a cost. This class is provided for compatibility with client Smalltalk standards. If you anticipate a large number of elements for your Set, we recommend you use the class IdentitySet. IdentitySet have faster access and can more easily handle constrained variables.

4.3 Stream Classes

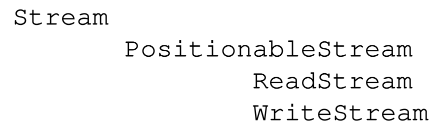
Reading or writing a SequenceableCollection's elements in sequence often entails some drudgery. At a minimum, you need to maintain an index variable so that you can keep track of which element you last processed.

Class Stream and its subclasses relieve you of this burden by simulating SequenceableCollections with more desirable behavior; a Stream acts like a SequenceableCollection that keeps track of the index most recently accessed. A

Stream that provides this kind of civilized access to a particular SequenceableCollection is said to “stream on” or “stream over” that collection.

There are two concrete Stream classes. Class ReadStream is specialized for reading SequenceableCollections and class WriteStream for writing them. These concrete Stream classes share two abstract superclasses, PositionableStream and Stream (see Figure 4.4).

Figure 4.4 Stream Class Hierarchy



```
Stream
    PositionableStream
        ReadStream
        WriteStream
```

This unusual juxtaposition of two abstract classes, Stream and PositionableStream, leaves an opening for you in the hierarchy in case you should ever decide to create a nonpositionable stream class for accessing, say, a LinkedList class of your own devising.

A stream provide its special kind of access to a collection by keeping two instance variables, one of which refers to the collection you wish to read or write, and the other to a position (an index) that determines which element is to be read or written next. A stream automatically updates its position variable each time you use one of Stream’s accessing messages to read or write an element.

Stream Protocol

Streams provide messages to write or read an element at the next position beyond the current position, change the current position without accessing any elements, and peek at the next element beyond the current one without changing the Stream's notion of its current position. Stream also provide messages to test for an empty collection and for the end of a stream. Finally, there is a message that returns the collection associated with a stream. Example 4.52 demonstrates the effect of several of these messages on a ReadStream.

Example 4.52

```
| aReadStream anArray |
anArray := #('item1' 'item2' 'item3' 'item4' 'item5').
aReadStream := ReadStream on: anArray.
UserGlobals at: #aReadStream put: aReadStream.
aReadStream position.        "What's the initial position?"
%
 1

"Return the item at the current position."
aReadStream next.
%
 item1

aReadStream position: 2.   "Set the position to the second
element"
aReadStream next.          "Read that element."
%
 item2

"Move to position 6. If at the end, reset the position to
the Stream's beginning"
aReadStream position: 6.     "Move past the last element"
(aReadStream atEnd)ifTrue:[aReadStream reset].
aReadStream next
%
 item1
```

Here is an example showing use of WriteStream:

Example 4.53

```
| aWriteStream |
aWriteStream := WriteStream on: (Array new: 5).
aWriteStream nextPut: 'item1'; nextPut: 'item2'.
UserGlobals at: #aWriteStream put: aWriteStream.
%
"Examine the Stream's contents"
aWriteStream contents
   verifyElementsIn: #( 'item1' 'item2' )
%
aWriteStream contents.
 an Array
 #1 item1
 #2 item2
 #3 nil
 #4 nil
 #5 nil

aWriteStream position: 4.
aWriteStream nextPut: 'item4'. "Store new item there."
aWriteStream nextPut: 'item5'. "Store item at next slot."
aWriteStream position: 1.      "Move to position 1."
"Replace item there."
aWriteStream nextPut: 'A new item at the front'.

"Examine the Stream's contents"
aWriteStream contents verifyElementsIn:
   #( 'A new item at the front' )
%
aWriteStream.itsCollection.
 an Array
 #1 A new item at the front
 #2 item2
 #3 nil
 #4 item4
 #5 item5
```

Creating Printable Strings with Streams

Streams are especially useful for building printable strings.

Example 4.54

```
| aStream aSet lineNumber |
lineNumber := 1.
aStream := WriteStream on: (String new).
aSet := IdentitySet withAll: #( 'lemur' 'gibbon' 'potto'
'siamang' 'rhesus' 'macaque' 'orangutan').
aSet do: [:i | aStream nextPutAll: lineNumber asString.
   aStream nextPutAll: ' '.
   aStream nextPutAll: i.
   aStream nextPut: Character lf.
   lineNumber := lineNumber + 1.  ].
aStream.itsCollection
%
aStream contents
 1 lemur
 2 gibbon
 3 potto
 4 siamang
 5 rhesus
 6 macaque
 7 orangutan
```

Chapter 5

Querying

This chapter describes Smalltalk's indexed associative access mechanism, a system for efficiently retrieving elements of large collections.

Relations
reviews the concept of relations.

Selection Blocks and Selections
describes how to use a path to select all the elements of a collection that meet certain criteria.

Additional Query Protocol
explains how to select a single element of a collection that meets certain criteria, or all those elements that do not meet them.

Indexing for Faster Access
discusses Smalltalk's facilities for creating and maintaining indexes on collections.

Nil Values and Selection
discusses the ramifications of using a path, one of whose elements might contain nil.

Paths Containing Collections
explains how you can use a path, one of whose elements is a collection instead of a single object.

Sorting and Indexing
describes protocol for sorting collections efficiently.

5.1 Relations

It's common practice to construct a relational database as a set of multiple-field records. Usually, each record represents one entity and each field in a record stores a piece of information about that entity. In a relational database, the set of records is called a relation, individual records are called tuples, and the fields are called attributes.

For example, the following table depicts a small relation that stores data about employees:

Figure 5.1 Employee Relation

Employees			
Name	Job	Age	Address
Fred	clerk	40	221 S. Main
Lurleen	busdriver	24	540 E. Sixth
Conan	librarian	40	999 W. West

In GemStone, it's natural to represent such a relation as an IdentityBag or IdentitySet of objects of class Employee, with each employee containing the instance variables *name*, *job*, *age*, and *address*. Each element of the IdentitySet corresponds to a record, and each instance variable of an element corresponds to a field.

To make it easy to retrieve values from a record, you can define selectors in class Employee so that an instance of Employee returns the value of its *name* instance variable when it receives the message `name`, the value of its *age* variable when it receives the message `age`, and so on. The discussion of class IdentitySet in Chapter 4, "Collection and Stream Classes," describes one way to develop this Employee class.

As that chapter also explains, you can use Collection's searching protocol to search for a record (element) containing a particular field (instance variable) value.

```
myEmployees select: [:anEmployee | anEmployee age = 40]
```

Searching for an object by content or value instead of by name or location is called *associative access*.

The searching messages defined by Collection must send one or more messages for each element of the receiver. Executing the expression given above requires sending the messages `age` and = for each element of myEmployees. This strategy is suitable for small collections, but it can be too slow for a collection containing thousands of complex elements.

For efficient associative access to large collections, it's useful to build an external index for them. Indexing a Collection creates structures such as balanced trees that let you find values without waiting for sequential searches. Indexing structures can retrieve the objects you require by sending many fewer messages—ideally, only the minimum number necessary. Indexes allow you faster access to large UnorderedCollections because when such collections are indexed, they can respond to queries using `select:`, `detect:`, or `reject:` without sending messages for every element of the receiver.

What You Need To Know

To use Smalltalk's facilities for searching large collections quickly, you need to:

1. Specify which of the instance variables in a collection's elements are indexed, using protocol from UnorderedCollection together with a special syntactic structure called a *path* to designate variables for indexing.
2. Construct a *selection block* whose expressions describe the values to be sought among the instance variables within the elements of a collection: when a selection block appears as the argument to one of UnorderedCollection's enumeration methods `select:`, `reject:`, and `detect:`, the method uses the indexing structures you've specified to retrieve elements quickly.

For example, if you planned to retrieve employees with certain jobs quickly and frequently, you need to create an "Employees" set that is indexed for fast associative access and then build an index on the *job* instance variable in each element of Employees. Then, to retrieve employees with a certain job, you build a selection block specifying the instance variable *job* and the target job, and send `select:` to Employees with the selection block as its argument.

Although you do not need to constrain the elements of a collection in order to index it, nor constrain the instance variables of its elements in order to search a

path they define, searching large collections goes faster if such constraints are specified. On the other hand, if such constraints are not specified, then you can search collections of heterogeneous elements, such as those containing several different versions of a class.

This chapter tells you how to specify indexes and perform selections, and it also provides some miscellaneous information to help you use those mechanisms efficiently.

5.2 Selection Blocks and Selection

Once you've created a collection, you can efficiently retrieve selected elements of the collection by formulating queries as enumeration messages that take selection blocks as their arguments.

A *selection block* is a syntactic variant of an ordinary Smalltalk block. When a collection receives `select:`, `detect:`, `reject:`, or one of several related messages, with a selection block as the argument, it retrieves those of its elements that meet the criteria specified in the selection block.

The following statement returns all Employees named 'Fred'. The selection block is the expression delimited by curly braces { }.

Example 5.1

```
|fredEmps |
fredEmps := myEmployees select:
          {:anEmployee | anEmployee.name = 'Fred'}.
```

This statement is similar to an example given earlier, in which `select:` took an ordinary block as its argument:

Example 5.2

```
fredEmps := myEmployees select:
          [:anEmployee | anEmployee.name = 'Fred'].
```

While square brackets `[]` delimit an ordinary block, curly braces `{}` delimit a selection block; Otherwise, the two statements look the same. A query using a selection block also returns the same results as if the selection block predicate had

been treated as a series of message expressions. However, some special restrictions apply to the query language.

Subsequent sections of this chapter describe selection block anatomy and behavior in general, and the query language restrictions in particular.

Selection Block Predicates and Free Variables

Like an ordinary, one-argument block, a selection block has two parts: the *free variable* and the *predicate*. In the following selection block, the free variable is to the left of the vertical bar and the predicate is to the right.

Figure 5.2 Anatomy of a Selection Block

```
fredEmps := myEmployees select:
 {:anEmployee | anEmployee.name = 'Fred'}
```


A free variable for the selection block is analogous to an argument for an ordinary block. As `select:` goes through myEmployees, it makes the free variable *anEmployee* represent each element in turn. In contrast to an ordinary block, which may have several arguments, a selection block can have only one free variable.

The predicate for a selection block is analogous to the right side of an ordinary block, which contains Smalltalk statements. In a selection block, the predicate must be a Boolean expression; usually, the expression compares an instance variable from among the objects to be searched with another instance variable or with a constant. In the example, for each element of the collection myEmployees, the predicate compares the element's instance variable *name* with the string 'Fred'.

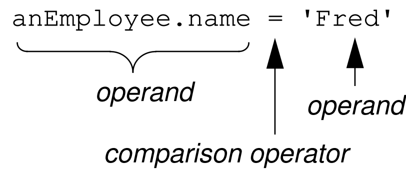
The method for `select:` gathers into the collection fredEmps each element whose *name* value makes the predicate true.

A predicate contains one or more *terms*—the expressions that specify comparisons.

Predicate Terms

A *term* is a Boolean expression containing an operand and usually a comparison operator followed by another operand, as shown in Figure 5.3:

Figure 5.3 Anatomy of a Selection Block Predicate Term



Predicate Operands

An operand can be a path (*anEmployee.name*, in this case), a variable name, or a literal ('Fred', in this example). All kinds of Smalltalk literals except arrays are acceptable as operands.

If a path points to objects within elements of `select:`'s receiver (as does *anEmployee.name*), then each variable in the path must be a valid instance variable name for the receiver and its elements. In this case, *anEmployee.name* is acceptable because the receiver holds employees and class Employee defines the instance variable *name*. The kind of constraint required on the last variable in such a path depends upon the kind of query in which the path is used.

Predicate Operators

Table 5.1 lists the comparison operators used in a selection block predicate:

Table 5.1 Comparison Operators Allowed in a Selection Block

Operator	Description
==	Identity comparison operator
=	Equality comparison operator, case-sensitive
<	Less than equality operator, case-insensitive
<=	Less than or equal to equality operator, case-insensitive
>	Greater than equality operator, case-insensitive
>=	Greater than or equal to equality operator, case-insensitive

No other operators are permitted in a selection block.

The associative query mechanism and Smalltalk do not follow exactly the same rules in determining the legality of comparisons. As in ordinary Smalltalk expressions, an identity comparison can be performed between two objects of any kind. The following peculiar query, for example, is perfectly legal:

Example 5.3

```
| aDateTime |
aDateTime := DateTime now.
myEmployees select: {:i | aDateTime == i.name}
```

However, not all kinds of objects are comparable using the equality operators =, <=, <, >=, >. If GemStone kernel classes are being compared, both operands must be of the same class, unless they are instances of String, Number, or DateTime. In that case, an operand can be an instance of a *subclass* of String, Number, or DateTime and still compare successfully with a String, Number, or DateTime, respectively. That is, you can use the equality operators in comparing any kind of String to any other kind of String, or any number to any other kind of number.

The following query, for example, results in an error because *age* and 'Frank' are of different classes.

```
myEmployees select: {:i | i.age <= 'Frank'}
```

The following query succeeds because Floats and Integers are both kinds of Number:

```
myEmployees select: {:i | 20.0 > i.age}
```

Because of its special significance as a placeholder for unknown or inapplicable values, nil is comparable to every kind of object in a selection block, and every kind of object is comparable to nil.

An attempt to execute a selection block that uses any of the equality operators including = to compare incomparable objects elicits an error notification.

Predicate Operators and User-defined Classes

If you need to, you can redefine the equality operators =, <=, <, >=, > in classes that you have created. In that case, the operands compared using these operators need not be of the same class. If you have created a class and redefined its equality operators, you can compare instances of that class with any other class that make sense for your application. See “Redefined Comparison Messages in Selection Blocks” on page 5-10 for further details.

Predicates Without Operators

A predicate can consist of a single Boolean path or variable. Suppose, for example, that Employee defined a Boolean variable named *isPermanent*. The following query returns all Employees in which *isPermanent* has the value true:

```
myEmployees select: {:i | i.isPermanent}
```

This query is equivalent to:

```
myEmployees select: {:i | i.isPermanent == true}
```

Conjunction of Predicate Terms

If you want retrieval of an element to be contingent on the values of two or more of its instance variables, you can join several terms using a conjunction (logical AND) operator. The conjunction operator, `&`, makes the predicate true if and only if the terms it connects are true.

The predicate in the following selection block is true for the Employees who are named Conan and work as librarians:

Example 5.4

```
| mySet |
mySet := myEmployees select: { :anEmployee |
 (anEmployee.name = 'Conan') & (anEmployee.job ='librarian')
}
```

This example returns a collection of the employees who meet the name and job criteria. Each conjoined term must be parenthesized.

You can conjoin as many as nine terms in a selection block.

If you do not wish to use the Boolean AND operator, but instead would like to learn which objects meet either one criterion OR another criterion, you must create two selection blocks, each querying about one of the criteria, and then merge the two resulting collections using the + operator for Set unions.

Example 5.5 shows how you can get a collection of all employees named either Fred or Ted.

Example 5.5

```
| fredsAndTeds freds teds |
freds := myEmployees select: { :anEmployee | anEmployee.name = 'Fred' }.
teds := myEmployees select: { :anEmployee | anEmployee.name = 'Ted' }.
fredsAndTeds := freds + teds
```

Limits on String Comparisons

In comparisons involving instances of String or its subclasses, the associative access mechanism considers only the first 900 characters of each operand. Two strings that differ only beginning at the 901st character are considered equal.

Redefined Comparison Messages in Selection Blocks

Because Smalltalk does not execute selection block predicates by passing messages to GemStone kernel classes, you cannot change the operation of a selection block by redefining the comparison messages in a GemStone kernel class. In other words, for predefined GemStone classes, the comparison operators really are operators in the traditional programming language sense; they are not messages.

For example, if you recompiled the class DateTime, redefining < to count backwards from the end of the century, Smalltalk would ignore that redefinition when < appeared next to an instance of DateTime inside a selection block. Smalltalk would simply apply an operator that behaved like DateTime's standard definition of <.

For subclasses that you have created, however, equality operators can be redefined. If you do so, the selection block in which they are used performs the comparison on the basis of your redefined operators—as long as one of the operands is the class you created and in which you redefined the equality operator.

If you redefine any, you must redefine at least the operators =, >, <, and <=. You can redefine one or more of these in terms of another if you wish.

The operators must be defined to conform to the following rules:

- If a < b and b < c, then a < c.
- Exactly one of these is true: a < b, *or* b < a, *or* a = b.
- a <= b if a < b or a = b.
- If a = b, then b = a.
- If a < b, then b > a.
- If a >= b, then b <= a.

You must obey one other rule as well: objects that are equal to each other must have equal hash values. Therefore, if you redefine =, you must also redefine the method `hash` to preserve this constraint between = and `hash`. Otherwise, dictionaries will not behave in a consistent and logical manner.

Suppose that you define the class Soldier as follows:

Example 5.6

```
Object subclass: #Soldier
       instVarNames: #( rank )
       classVars:  #( #Ranks )
       poolDictionaries: #()
       inDictionary: UserGlobals
       constraints: #[ #[ #rank, Symbol ] ]
       isInvariant: false

Soldier compileAccessingMethodsFor: Soldier.instVarNames

method: Soldier
hash
   "Answer a hash value based on the receiver's rank, since
    equality is defined in terms of a Soldier's rank. "
^ rank hash
%
```

You then compile accessing methods for its instance variables, and define an initialization method to initialize the class variable *Ranks*, as in the following example:

Example 5.7

```
classmethod: Soldier
initialize
 "Initialize the class variable Ranks as an array."
 | index |
Ranks := SymbolKeyValueDictionary new.
index := 1.
#( #Lieutenant #Captain #Major #Colonel #General )
      do: [:e | Ranks at: e put: index.
            index := index + 1 ].
%
Soldier initialize
```

We then initialize the class by executing the expression:

```
Soldier initialize
```

We define the equality operators in the class Soldier as follows:

Example 5.8

```
method: Soldier
 < aSoldier
"Return true if the rank of the receiver is lower than the
rank of the argument.  Return false otherwise, or if
either receiver or argument is unranked."
^ (Ranks at: rank otherwise: (Ranks size + 1)) <
      (Ranks at: aSoldier rank otherwise: 0)
%
method: Soldier
= aSoldier
"Return true if the rank of the receiver is equal to the
rank of the argument.  Return false otherwise, or if
either receiver or argument is unranked."
^ (Ranks at: rank otherwise: -1) =
      (Ranks at: aSoldier rank otherwise: 0)
%
method: Soldier
> aSoldier
"Greater than is defined in terms of less than."
^ aSoldier < self
%
method: Soldier
<= aSoldier
 "Return true if the rank of the receiver is less than or
equal to the rank of the argument.  Return false
otherwise, or if either receiver or argument is unranked."
^ (Ranks at: rank otherwise: (Ranks size + 1) <=
      (Ranks at: aSoldier rank otherwise: 0))
%
method: Soldier
hash
"Return a hash value based on the receiver's rank, because
equality is defined in terms of a Soldier's rank."
^ rank hash
%
```

We now create instances of Soldier having each possible rank, naming them aLieutenant and so on. We also create an instance of Soldier without any rank, and name it aPrivate:

Example 5.9

```
| tmp myArmy tmp2 |
myArmy := IdentityBag new.
1 to: 5 do: [:i |
  tmp := (Soldier.classVars at: #Ranks) keys do: [:tmp |
    tmp2 := (Soldier new rank: tmp).
    UserGlobals at: ('a' + tmp) asSymbol put: tmp2.
    myArmy add: tmp2
    ].
  ].
tmp2 := (Soldier new rank: #Private).
UserGlobals at: #aPrivate put: tmp2;
   at: #myArmy put: (myArmy add: tmp2; yourself) .
^ myArmy
%
```

We can now execute expressions of the form:

Example 5.10

```
aLieutenant < aMajor
true

aCaptain < aLieutenant
false
```

Expressions in selection blocks get the same results. Given a collection of soldiers named myArmy, the following selection block collects all the officers:

Example 5.11

```
| officers |
officers := myArmy select: { :aSoldier | aSoldier > aPrivate }
```

Changing the Ordering of Instances

Once you redefine the equality operators for a given class and create instances of that class, your instances may not remain the same forever. For example, the soldiers we created in Example 5.9 above may not all stay the same rank for their entire careers. Some may be promoted; others may be demoted. If an instance of Soldier changes its ordering relative to the other instances, you must manually update the equality index in which it participates. Because you have redefined the equality operators, GemStone has no way of determining how to update the index automatically, as it will when you use the system-supplied equality operators.

To handle updating the equality index in your application, follow these steps:

1. Confine code that can change the relative ordering of instances to as few places as possible. For the class Soldier, for example, we would write two methods: `promoteTo:` and `demoteTo:`. Code that changed the relative ranking of soldiers would appear only within these two methods.

2. Before the code that changes the ordering of the instance, include a line such as the following:

   ```
   anArray := self removeObjectFromBtrees
   ```

   The method `removeObjectFromBtrees` returns an array that you will need later within the method. Therefore you must assign the result to some variable—`anArray` in the example above.

3. After the code that changes the ordering of the instance, include a line such as the following:

   ```
   self addObjectToBtreesWithValues: anArray
   ```

If the ordering of the instance depends on more than one instance variable, this pair of lines must appear in the methods that set the value of each instance variable.

CAUTION:
Failing to include these lines can corrupt the equality index and lead to your application receiving GemStone errors notifying you that certain objects do not exist. Removing and re-creating the equality index may not fix the problem.

Collections Returned by Selection

The message `select:` returns a collection of the same class as the message's receiver. For example, sending `select:` to a SetOfEmployees results in a new SetOfEmployees.

NOTE:
When sent to an instance of RcIdentityBag, the message `select:` *returns an instance of IdentityBag instead. This is because the reduced-conflict classes use more memory or disk space than their ordinary counterparts, and conflict is not ordinarily a problem with collections returned from a query. If it causes a problem for your application, however, you can convert the resulting instance* `myBag` *to an instance of RcIdentityBag with an expression such as either of the two following:*

```
RcIdentityBag withAll: myBag
RcIdentyfBag new addAll: myBag
```

See "Transactions and Concurrency Control" on page 6-1 for further details on class RcIdentityBag.

The collection returned by a selection query has no index structures (the following sections of this chapter describe indexes). This is because indexes are built on individual instances of nonsequenceable collections rather than the classes. If you want to perform indexed selections on the new collection, you must build all of the necessary indexes. A later section, "Transferring Indexes" on page 5-26, describes a technique for duplicating a collection's indexes in a new instance.

Streams Returned by Selection

The result of a selection block can be returned as a stream instead of a collection. Returning the result as a stream is faster. If you are not sure that your query is precisely the right one, using a stream allows you to test the results with minimal overhead.

When GemStone returns the result of a selection block as a collection, the following operations must occur:

1. Each object in the result must be read.
2. The collection must be created.
3. Each object in the result must be put into the collection.

For a collection consisting of 10,000 objects, these operations can take a significant amount of time. By contrast, when GemStone returns the result of a selection block as a stream, the resulting objects are returned one at a time. Each object you request is read once, resulting in significantly faster performance and less overhead.

Streams do not automatically save the resulting objects. If you do not save them as you read them, the results of the query are lost.

The results of a selection block can be returned as a stream using the method `selectAsStream:`. This method returns an instance of the class RangeIndexReadStream, similar to a ReadStream but making more efficient use of GemStone resources. Like instances of ReadStream instances of RangeIndexReadStream understand the messages `next` and `atEnd`.

Suppose your company wishes to send a congratulatory letter to anyone who has worked there for ten years or more. Once you have sent the letter, you have no further use for the data. Assuming that each employee has an instance variable called *lengthOfService*, you can use a stream to formulate the query as follows:

Example 5.12

```
method: Employee
sendCongratulations
^ 'Congratulations. Thank you for your years of service. '
%
myEmployees createEqualityIndexOn: 'lengthOfService'
   withLastElementClass: SmallInteger

| oldTimers anOldTimer |
oldTimers := myEmployees selectAsStream:
          { :anEmp | anEmp.lengthOfService >= 10}.
[ oldTimers atEnd ] whileFalse: [
   anOldTimer := oldTimers next.
   anOldTimer sendCongratulations.  ].
%
nil
```

The method `selectAsStream:` has certain limitations, however.

- It takes a single predicate only; no conjunction of predicate terms is allowed.
- The collection you are searching must have an equality index on the path specified in the predicate. (Creating equality indexes is discussed in the section "Indexing For Faster Access" on page 5-20.)
- The predicate can contain only one path.

For example, the predicate shown in the following compares the result of one path with the result of another and therefore cannot be used with `selectAsStream:`

Example 5.13

```
myEmployees select: { :emp | emp.age > emp.lengthOfService }

myEmployees  createEqualityIndexOn: 'age'
            withLastElementClass: SmallInteger;
   createEqualityIndexOn: 'lengthOfService'
            withLastElementClass: SmallInteger

myEmployees selectAsStream:
     { :emp | emp.age > emp.lengthOfService }
%
```

Formulating a query using `selectAsStream:` is inappropriate for these cases:

- You wish to modify the receiver of the message (the nonsequenceable collection) by adding or removing elements.
- You want to modify instance variables upon which the query is based in the elements returned by the stream, while you are accessing the stream. Doing so can cause a GemStone error or corrupt the stream. If you must modify the receiver or its elements based on the query, use `select:` instead, which returns the entire resulting collection at once.

5.3 Additional Query Protocol

In addition to `select:`, three other messages search when sent to a collection with a selection block as argument.

If you want to use the associative access mechanism to retrieve all elements of a constrained Collection for which a a selection block is false, send `reject:` *aBlock*. The following expression, for example, retrieves all elements of myEmployees not named 'Lurleen':

Example 5.14

```
myEmployees reject: {:i | i.name = 'Lurleen'}
```

The messages `detect:` *aBlock* and `detect:` *aBlock* `ifNone:` *exceptionBlock* can also take selection blocks as arguments when sent to collections. The message `detect:` *aBlock* returns a single element of the receiver that meets the criteria specified in *aBlock*. The following expression returns an Employee of age 40:

Example 5.15

```
myEmployees detect: {:i | i.age = 40}
```

Since nonsequenceable collections are by definition unordered, there is no telling which element will be returned when there are several qualified candidates. If no elements are qualified, `detect:` issues an error notification and the interpreter halts.

If you don't want the interpreter to halt in the event of a fruitless search, use `detect:` *aBlock* `ifNone:` *exceptionBlock*. See Chapter 4, "Collection and Stream Classes.")

5.4 Indexing For Faster Access

Although queries using selection blocks can execute more rapidly than conventional selections that pass messages, their default behavior is to search collections in a relatively inefficient sequential manner. Given the right information, however, Smalltalk can build indexes that use as keys the values of instance variables within the elements of a collection. The keys can be the collection's elements or the values of instance variables of the collection's elements. In fact, keys can be the values of variables nested within the elements of a collection up to 16 levels deep. Values that serve as keys need not be unique.

In the presence of indexes, collections need not be searched sequentially in order to answer queries. Therefore, searching a large indexed collection can be significantly faster than searching a similar, nonindexed collection.

Smalltalk can create and maintain two kinds of indexes: *identity indexes*, which facilitate identity queries, and *equality indexes*, which facilitate equality queries.

Identity Indexes

Identity indexes accelerate identity queries. The simplest kind of identity query selects the elements of a collection in which some instance variable is identical to (or not identical to) a target value. The following example retrieves from a collection of employees those elements in which the instance variable *age* has the value 40:

Example 5.16

```
|age40Employees |
age40Employees := myEmployees select:
 {:anEmployee | anEmployee.age == 40}
aSetOfEmployees
```

In order to execute such a query with the greatest possible efficiency, you need to have built an identity index on the constrained path to the instance variable *age*.

Creating Identity Indexes

To create an identity index, use UnorderedCollection's selector `createIdentityIndexOn:`, which takes as its argument a path, specified as a string. Here is a message telling myEmployees to create an identity index on the instance variable *age* within each of its elements:

```
myEmployees createIdentityIndexOn: 'age'.
```

Another example may be helpful. Given that each Employee's instance variable *address* contains another instance variable, *zipcode*, the following statement creates an identity index on the zipcodes nested within the elements of the IdentityBag MyEmployees:

```
myEmployees createIdentityIndexOn: 'address.zipcode'.
```

For large collections, it may take a long time to create an index in one transaction. So, for these large collections, you may choose to commit your work to the repository before the index is completed with the method:

`createIdentityIndexOn:` *`aPathString`* `commitInterval:` *`aNumber`*

The sender specifies an interval what which to commit results during the enumeration of the collection. By breaking the index creation into multiple, smaller transactions, the overall time required to build the index is shorter.

While the index is being created, the index is write-locked. Any query that would normally use the index is performed directly on the collection, by brute force. If a concurrent user modifies a object that is actively participating in the index at the same time, the `createIdentityIndexOn:` method is terminated with an error.

The message `progressOfIndexCreation` returns a description of the current status for an index as it is created.

Equality Indexes

Equality indexes facilitate equality queries. The simplest kind of equality query selects the elements of a collection in which a particular named instance variable is equal to some target value.

The following example retrieves from a collection of employees those elements in which the instance variable *name* has the value 'Fred':

Example 5.17

```
| freds |
freds := myEmployees select:
  { :anEmployee | anEmployee.name = 'Fred' }
aSetOfEmployees
```

As explained in a previous section, equality queries use the related comparison operators =, <, <=, >, and >=.

You can create equality indexes on the following kinds of objects:

- Boolean
- Character
- DateTime
- Number
- String
- UndefinedObject

You can create equality indexes on classes you have defined, as long as they either implement or inherit at least methods for the selectors =, >, >=, <, <=. One or more of these methods can be implemented in terms of the others, if necessary.

Creating Equality Indexes

The technique for creating equality indexes is similar to the technique for creating identity indexes. To create an equality index for a path whose instance variables are all constrained, send the message `createEqualityIndexOn:` with a path as its argument.

You can create an equality index on the instance variable *name* within each element of the collection myEmployees, assuming that the class of myEmployees is constrained to hold instances of the class Employee, and *name* is constrained to be an instance of String:

```
myEmployees createEqualityIndexOn: 'name'.
```

To create an equality index directly on the elements of a collection instead of on instance variables of those elements, you can give an empty string as the path argument to the indexing message. The following example creates an equality index on the elements of the collection aBagOfAnimals (a collection constrained to hold only instances of Animal):

```
aBagOfAnimals createEqualityIndexOn: ''.
```

You can also create an equality index on the elements of a collection, or on instance variables of those elements, when any or all elements of the path, including the final one, have not been constrained. To do so, specify the class of the final element of the path by sending the message:

```
createEqualityIndexOn: aPath withLastElementClass: aClass
```

The argument to the first keyword is a path (or an empty string); the argument to the second keyword is the name of the class whose instances you expect to encounter at the end of the path.

Some examples:

```
aBagOfAnimals createEqualityIndexOn: ''
      withLastElementClass: Animal.

myEmployees createEqualityIndexOn: 'address'
            withLastElementClass: Address.

myEmployees createEqualityIndexOn: 'department.manager'
         withLastElementClass: Employee.
```

Creating Indexes on Very Large Collections

For large collections, it may take a long time to create an index in one transaction. For those collections, you may choose to commit your work to the repository before the index is completed with the methods:

```
createEqualityIndexOn: aPathString commitInterval: aNumber
createEqualityIndexOn: aPathString withLastELementClass: aClass
  commInterval: aNumber
```

These messages work in the manner previously described, but the sender specifies an interval what which to commit results during the enumeration of the collection. By breaking the index creation into multiple, smaller transactions, the overall time required to build the index is shorter.

While the index is being created, the index is write-locked. Any query that would normally use the index is performed directly on the collection, by brute force. If a concurrent user modifies a object that is actively participating in the index at the same time, the method terminates with an error.

The message `progressOfIndexCreation` returns a description of the current status for an index as it is created.

Automatic Identity Indexing

Smalltalk can build either identity or equality indexes on atomic objects—that is, instances of Boolean, Character, SmallInteger and UndefinedObject. In fact, for those kinds of objects, equality and identity are the same, so creating an equality index effectively creates an identity index as well.

Implicit Indexes

In the process of creating an index on a nested instance variable, Smalltalk also creates identity indexes on the values that lie on the path to that variable. For example, creating an equality index on *last* in the following expression also creates an identity index on *name*.

```
myEmployees createEqualityIndexOn: 'name.last'.
```

Therefore, executing the above expression allows you to make indexed identity queries in terms of *name* values without explicitly creating an index on *name*.

Indexes and Transactions

Adding an object to an indexed collection writes that object to the repository. Similarly, creating or removing an index on a collection writes elements of the collection. Finally, modifying an object that participates in an index on some collection can, under certain circumstances, write certain objects built and maintained internally by GemStone as part of the indexing mechanism. Chapter 6, “Transactions and Concurrency Control,” explains the significance of your writing an object.

Inquiring About Indexes

Class UnorderedCollection defines messages that enable you to ask collections about the indexes on their contents. These messages are:

- `equalityIndexedPaths` and `identityIndexedPaths`

  Returns, respectively, the equality indexes and the identity indexes on the receiver's contents. Each message returns an array of strings representing the paths in question.

  This example returns the paths into myEmployees that bear equality indexes:

  ```
  myEmployees equalityIndexedPaths
  ```

- `kindsOfIndexOn:` *aPathNameString*

  Returns information about the kind of index present on an instance variable within the elements of the receiver. The information is returned as one of these symbols: #none, #identity, #equality, #identityAndEquality.

- `equalityIndexedPathsAndConstraints`

  Returns an array in which the odd-numbered elements are the elements of the path, and the even-numbered elements are the constraints, if any, on those elements. These include both constraints specified explicitly in a class definition and those specified when creating an index using the keyword `withLastElementClass:`.

The following sections describe several practical uses for these messages.

Removing Indexes

Class UnorderedCollection defines these messages for removing indexes from a collection:

- `removeEqualityIndexOn:` *aPathString*

  Removes an equality index from the variable indicated by *aPathString*. If the path specified does not exist (perhaps because you mistyped), this method returns an error. If the path specified was implicitly created, the method returns the path, but the index is not removed. If the index is successfully removed, the method returns the receiver.

- `removeIdentityIndexOn:` *aPathString*

  Removes identity indexes. If the path specified does not exist, the method returns an error. If the path specified was implicitly created, the method returns the path, but the index is not removed. If the index is successfully removed, the method returns the receiver.

- `removeAllIndexes`

  Removes all explicitly created indexes from the receiver. If the receiver retains implicit indexes after the removal, this method returns an array indicating that the receiver participates, as an element of a path, in indexes created on other collections. Otherwise, this method returns the receiver.

For complete information on these methods, see the *GemStone Kernel Reference*.

Implicit Index Removal

As previously explained, building an index on the path 'a.b.c' causes GemStone to create implicit identity indexes on the paths 'a.b' and 'a', as well. When you remove explicitly created indexes, the implicit ones that were created on the same path are also removed. That is, when you remove indexes from the path 'a.b.c', GemStone also removes the implicit indexes from the paths 'a.b' and 'a'.

Implicitly created indexes cannot be explicitly removed. However, explicitly created indexes must be explicitly removed.

Transferring Indexes

As explained elsewhere in this chapter, a collection returned by `select:` is devoid of indexing, even when `select:`'s receiver has indexes in place.

In general, assume that any operation that transfers or re-creates the elements of a collection preserves any constraints it might have, but destroys indexes. For

example, when you copy a collection, the copy retains the original constraints but loses indexes. Migrating an indexed collection destroys its indexes.

Fortunately, the index inquiry protocol for UnorderedCollection makes it easy to transfer indexes to a new collection:

Example 5.18

```
| someEmployees identityIndexes equalityIndexes |
"First, gather some elements of myEmployees into a new
Collection."
someEmployees := myEmployees select:
                 { :anEmp | anEmp.job = 'clerk'}.
"Now make some arrays containing the indexes on employees."
identityIndexes := myEmployees identityIndexedPaths.
equalityIndexes := myEmployees equalityIndexedPaths.

"For each index on myEmployees, create a similar index on
someEmpolyees."
1 to: (identityIndexes size) do:
     [ :n | someEmployees createIdentityIndexOn:
                    (identityIndexes at: n) ].
1 to: (equalityIndexes size) do:
     [ :n | someEmployees createEqualityIndexOn:
(equalityIndexes at:n )withLastElementClass: SmallInteger ].
```

Removing and Re-creating Indexes

For several reasons, you may sometimes wish to remove indexes temporarily and then create them again. For example, you may wish to accelerate updates or you may be migrating a class to a new version.

Whenever you change the value of an object that participates in an index, Smalltalk automatically adjusts the indexes to accommodate the new value. Obviously, this entails more work than must ordinarily be done when a value changes.

Therefore, when your program needs to make a large batch of changes to an object that participates in an index, it might be most efficient to remove some or all of the object's indexes before performing the updates. When the frequency of updates to the object decreases, you can rebuild the indexes to accelerate queries again.

When you migrate an explicitly indexed collection from one version to another, its index is automatically removed. To accelerate queries, you may wish to create the index anew after the collection has been migrated. If the original path was not constrained, an error arises if you use `equalityIndexedPaths` to determine the indexes beforehand so that you can re-create them later. Instead, send the message `equalityIndexedPathsAndConstraints` to determine the argument to use, and re-create the index with the message `createEqualityIndexOn: withLastElementClass:`, using as the argument to the last keyword the appropriate class returned from `equalityIndexedPathsAndConstraints`. See the comment for this method in the *GemStone Kernel Reference* for complete details on its return value.

Indexing and Authorization

When you query an UnorderedCollection that contains an element you are not authorized to read, you will get an authorization error, regardless of whether the collection is indexed, However, under certain narrow circumstances, indexing an UnorderedCollection can cause spurious authorization errors. The sequence of events that can lead to such errors is:

1. An object that participates in the index was once assigned to a segment for which you lack authorization.
2. The object is moved from that segment to another segment that you are, in fact, authorized to read.

Under those circumstances, the indexing system does not automatically recognize the move, which causes inappropriate authorization errors for queries on the UnorderedCollection. If you believe that you are experiencing spurious authorization errors, execute an expression of the form:

```
aBag recomputeIndexSegments
```

Executing such an expression removes the problem for the duration of your session. If you wish to remove the problem for other sessions or other users, commit the transaction after executing the expression.

Indexing and Performance

Under ordinary circumstances, indexing a large collection speeds up queries performed on that collection and has little effect on other operations. Under certain uncommon circumstances, however, indexing can cause a performance bottleneck.

For example, you may notice slower than acceptable performance if you are making a great many modifications to the instance variables of objects that participate in an index, and:

- the path of the index is long; or
- the object occurs many times within the indexed IdentityBag or Bag (recall that neither IdentitySet nor Set may have multiple occurrences of the same object); or
- the object participates in many indexes.

Even so, indexing a large collection is still likely to improve performance unless more than one of these circumstances holds true. If you do experience a performance problem, you can work around it in one of two ways:

- If you have created relatively few indexes but are modifying many indexed objects, it may be worthwhile to remove the indexes, modify the objects, and then re-create the indexes.
- If you are making many modifications to only a few objects, or if you have created a great many indexes, it is more efficient to commit frequently during the course of your work. That is, modify a few objects, commit the transaction, modify a few more objects, and commit again. Frequent commits improve performance noticeably.

Indexing Errors

When you create an index on an unconstrained collection, or on a path that includes unconstrained instance variables, it is possible to encounter an object for which the specified path is in error. For example, imagine that the class Employee defines the instance variable *address*, which is intended to store instances of the class Address. The current class Address includes an instance variable named *zipCode*. However, the employees that have worked for your company the longest store instances of a previous version of Address that did not include this instance variable. You then attempt to create an index on the following path for such a collection:

```
myEmployees createEqualityIndexOn: 'address.zipCode'
```

When GemStone finds the employees whose addresses do not contain a zip code, it notifies you of an error. However, creating an index is an operation that creates a complex and specialized indexing structure. An error in the middle of this operation can leave the indexing structure in an inconsistent state. In order to avoid this, a transaction in which such an operation occurs cannot be committed.

If you think you may have a collection in which this could be a problem, create its index in a transaction by itself.

For the same purpose of maintaining the internal consistency of indexing structures, authorization errors encountered while creating or removing an index, adding or removing an object from an indexed collection, or changing an object that participates in an index also prevent the transaction in which they occurred from being committed.

For details on authorization errors, see the *GemStone System Administration Guide*. For details on committing transactions, see Chapter 6.

Errors can also arise when you must remove indexes and re-create them later.

5.5 Nil Values and Selection

Be careful about storing nil in any instance variable that can participate in indexed queries. When nil appears anywhere in a path except as the last value, selection block predicates using that path return false without any warning that false might be a spurious result.

For example, suppose that each employee's *name* instance variable contained the two additional indexed variables *first* and *last*. Imagine, too, that some employees had *name* set to nil to represent "unknown." The following selection, meant to return all employees not named Sergio, also excludes those whose names are unknown (nil):

```
myEmployees select: {:anEmp | anEmp.name.first ~= 'Sergio'}
```

The result might be misleading, because employees with undefined names (nil) are excluded in addition to employees named 'Sergio'. If it is impractical to rule out the possibility of employees with nil names, then it is preferable to formulate the query this way:

```
myEmployees select: {:anEmp | (anEmp.name ~= nil) &
                        (anEmp.name.first ~= 'Sergio') }
```

Problems with misleading results can also arise if you compare NaN values during searches. For purposes of associative access, a NaN is never equal to anything else. This means that you cannot use `select:` to search for NaNs.

5.6 Paths Containing Collections

So far in this chapter, we have discussed only paths that traverse single instance variables from object to object. Another kind of path is created when any term except the first, includes an UnorderedCollection instead of a single object.

Paths that include an UnorderedCollection can be used within selection blocks, to create indexes, or to remove indexes. They cannot be used for sorting or to replace any other kind of expression.

When you wish to specify a path containing an UnorderedCollection, the collection is represented by an asterisk. For example, suppose you want to know which of your employees has children of age 18 or younger. To facilitate such queries, each of your employees has an instance variable named *children*, which is implemented as a set. This set contains instances of a class that has an instance variable named *age*. We can make the query this way:

```
myEmployees select: {:anEmp | anEmp.children.*.age <= 18 }
```

The asterisk in the path above indicates that the query is to examine all elements of the set *children*.

You are free to put as many asterisks in a path as you need, except as the first term in the path. However, a combinatorial explosion can result. A query searching all elements of a collection that in turn includes other collections is potentially time-consuming if the collections are large.

To take another example, suppose that each employee has a set of skills, and each skill is ranked at a level of 1, 2, or 3, where a skill level of 1 means that the employee is a novice at the skill and requires supervision, 2 means that the employee can perform work requiring the skill without supervision, and 3 means that the employee can supervise another employee trying to learn that skill. Each employee has an instance variable *skills*, which is implemented as a set. The set contains instances of class Skill, a relatively simple object containing the instance variable *type*—a string such as `'welding'` or `'lathing'`, and the instance variable *level*, which is an integer.

If you want to know which employees possessed some skills at level 3, and could therefore function as teachers, you can query the object server as follows:

```
myEmployees select: {:anEmp | anEmp.skills.*.level = 3 }
```

This query returns all employees who possess any skill at all at the highest level. If you wish to know which employees have reached level 3 at *all* their skills, you must subtract those employees who have any skills at a lower level from the

previous result. For this purpose, IdentityBag implements the selector – (minus) to return only the difference between the two sets:

Example 5.19

```
Object subclass: 'Skill'
  instVarNames: #('level' )
  classVars: #( )
  poolDictionaries: #[ ]
  inDictionary: UserGlobals
  constraints: #[ #[ #level, SmallInteger ] ]
  isInvariant: false

Skill compileAccessingMethodsFor: Skill.instVarNames

(IdentitySet subclass: 'SetOfSkills'
  instVarNames: #( )
  classVars: #( )
  poolDictionaries: #[ ]
  inDictionary: UserGlobals
  constraints: Skill
  isInvariant: false) name
(Object subclass: 'Employee'
  instVarNames: #('skills' 'name' 'job' 'age' 'address'
'salary')
  classVars: #( )
  poolDictionaries: #[ ]
  inDictionary: UserGlobals
  constraints: #[ #[ #skills, SetOfSkills ] ]
  isInvariant: false) name
Employee compileAccessingMethodsFor: Employee.instVarNames

(IdentitySet subclass: 'SetOfEmployees'
  instVarNames: #( )
  classVars: #( )
  poolDictionaries: #[ ]
  inDictionary: UserGlobals
  constraints: #( )
  isInvariant: false) name

| Conan Lurleen Fred myEmployees |
Conan := (Employee new) name: 'Conan'; job: 'librarian';
```

```
   age: 40; address: '999 W. West'; salary: 123.45;
   skills: (SetOfSkills new add:
            (Skill new level: 2);   yourself).
Fred := (Employee new) name: 'Fred'; job: 'clerk';
   age: 40; address: '221 S. Main';
   skills: (SetOfSkills new add:
            (Skill new level: 1); yourself).
Lurleen := (Employee new) name: 'Lurleen'; job: 'busdriver';
   age: 24; address: '540 E. Sixth';
   skills: (SetOfSkills new add:
            (Skill new level: 3); yourself).

myEmployees := SetOfEmployees new.
UserGlobals at: #myEmployees put: myEmployees;
   at: #Conan put: Conan; at: #Fred put: Fred;
   at: #Lurleen put: Lurleen.

myEmployees add: Fred; add: Lurleen; add: Conan; yourself

(myEmployees select: {:anEmp | anEmp.skills.*.level = 3 }) -
  (myEmployees select: {:anEmp | anEmp.skills.*.level < 3 })
```

If, on the other hand, you wish to know which employees have not reached level 3 at *any* skill, you must find all employees who have any skills at the lower levels, and then subtract those who also have one or more skills at level 3:

Example 5.20

```
(myEmployees select: {:anEmp | anEmp.skills.*.level < 3 }) -
  (myEmployees select: {:anEmp | anEmp.skills.*.level = 3 })
```

Using this functionality can produce one kind of surprise—you can occasionally get a result that has more elements than the receiver. For example, recall the query we used to determine which employees had children of age 18 or younger. If an employee in the IdentityBag myEmployees has two children who meet the criterion, that employee is represented in the result twice. Employees with three minor children appear in the result three times, and so on. You may find this result a bit startling, but it conforms to the intent of the class IdentityBag, which allows duplicate entries.

5.7 Sorting and Indexing

Although indexes are not necessary for sorting, Smalltalk can take advantage of equality indexes to accelerate some kinds of sorts. Specifically, an index is helpful in sorting on a path consisting of at most one instance variable name. For example, an equality index on *name* makes the following expression execute more quickly than it would in the absence of an index:

```
myEmployees sortAscending: 'name'
```

Similarly, the following expression sorts an IdentityBag more rapidly with an index on the path ' ' (the elements of the collection):

```
myBagOfStrings sortAscending: ''.
```

Chapter 6

Transactions and Concurrency Control

GemStone users can share code and data objects by maintaining common dictionaries that refer to those objects. However, if operations that modify shared objects are interleaved in any arbitrary order, inconsistencies can result. This chapter describes how GemStone manages concurrent sessions to prevent such inconsistencies.

Gemstone's Conflict Management
introduces the concept of a transaction and describes how it interacts with each user's view of the repository.

Changing Transaction Mode
describes how to start and commit, continue, or abort a transaction in either automatic or manual transaction mode.

Concurrency Management
introduces optimistic and pessimistic concurrency control.

Controlling Concurrent Access With Locks
discusses the kids of lock you can use to prevent conflict.

Classes That Reduce the Chance of Conflict
describes the classes that help reduce the likelihood of a conflict.

6.1 Gemstone's Conflict Management

GemStone prevents conflict between users by encapsulating each session's operations (computations, stores, and fetches) in units called *transactions*. The operations that make up a transaction act on what appears to you to be a private *view* of GemStone objects. When you tell GemStone to *commit* the current transaction, GemStone tries to merge the modified objects in your view with the shared object store.

Figure 6.1 illustrates how your view is updated.

Transactions

A transaction maintains a consistent view of the repository. Objects that the repository contained when you start a transaction are preserved in your view, even if you are not using them and other users' actions have rendered them obsolete. The storage that they are using cannot be reclaimed until you commit or abort your transaction. Depending upon the characteristics of your particular installation (such as the number of users, the frequency of transactions, and the extent of object sharing), this burden can be trivial or significant.

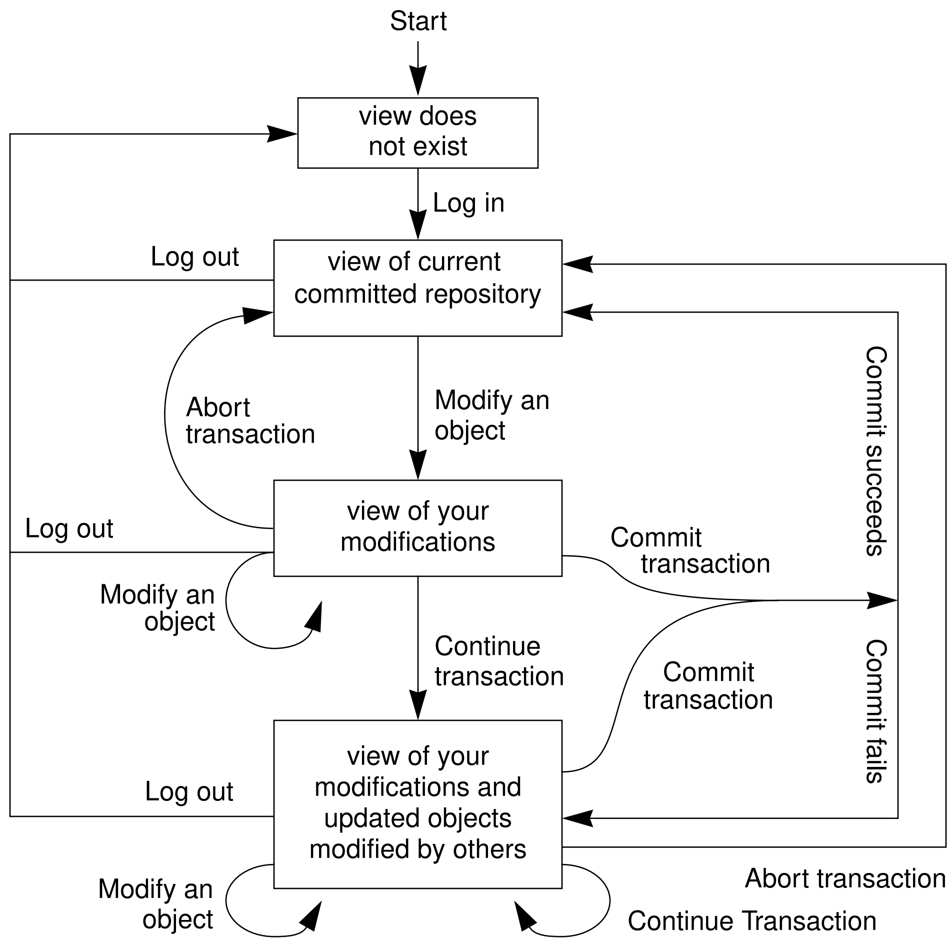
When you log into GemStone, a transaction is started for you. This transaction exists until you successfully commit the transaction, abort it, or log out. Your view endures for the length of the transaction, unless you explicitly choose to continue the transaction and get a new view.

When Should You Commit a Transaction?

Most applications create or modify objects in logically separate steps, combining trivial operations in sequences that ultimately do significant things. To protect other users from reading or using intermediate results, you want to commit after your program has produced some stable and usable results. Changes become visible to other users only after you've committed.

Your chance of being in conflict with other users increases with the time between commits.

Figure 6.1 View States



Reading and Writing in Transactions

GemStone considers the operations that take place in a transaction (or view) as *reading* or *writing* objects. Any operation that sends a message to an object, or accesses any instance variable of an object, is said to *read* that object. An operation that stores something in one of an object's instance variables is said to *write* the object. While you can read without writing, writing an object always implies

reading it. GemStone must read the internal state of an object in order to store a new value in the object.

Operations that fetch information about an object also read the object. In particular, fetching an object's size, class, or segment reads the object. An object also gets read in the process of being stored into another object.

The following expression sends a message to obtain the name of an employee and so reads the object:

```
theName := anEmployee name.           "reads anEmployee"
```

The following example, reads `aName` in the process of storing it in anEmployee. Reading `aName` is necessary to ensure that storing it won't violate any of anEmployee's constraints on the instance variable *name*:

```
anEmployee name: aName  "writes anEmployee, reads aName"
```

In this example, anEmployee is written in the same operation that `aName` is read.

Some less common operations cause objects to be read or written:

- Assigning an object to a new segment, using the message `assignToSegment:`, writes the object and reads both the old and the new segment.
- Adding an object to an indexed collection for the first time, creating an index on a collection, or removing an index on a collection writes the elements of the collection.
- Modifying an object that participates in an index may write support objects built and maintained as part of the indexing mechanism.

For the purposes of detecting conflict among concurrent users, GemStone keeps separate sets of the objects you have written during a transaction and the objects you have only read. These sets are called the *write set* and the *read set*; the read set is always a superset of the write set.

Reading and Writing Outside of Transactions

Outside of a transaction, reading an object is accomplished precisely the same way. You can write objects in the same way as well, but you cannot commit these changes to make them a permanent part of the repository.

6.2 How GemStone Detects Conflict

GemStone detects conflict by comparing your read and write sets with those of all other transactions committed since your transaction began. The following conditions signal a possible concurrency conflict:

- An object in your write set is also in the write set of another transaction—a *write/write conflict*. Write-write conflicts can involve only a single object.
- An object in your write set is in the read set of another transaction, and an object in your read set is in any other concurrent transaction's write set—a *read/write conflict*. A *concurrent transaction* is any transaction that was committed between the time you started your transaction and the time you tried to commit your transaction. The conflicting read set and the conflicting write set need not be associated with the same concurrent transaction. A read-write conflict must always involve at least two objects, but sometimes involves more.

You set the level of checking the object server does with the CONCURRENCY_MODE configuration parameter in your application's configuration file. Your choices are:

- FULL_CHECKS

  Checks for both write/write and read/write conflicts. This is the default mode because it is the safest.

- NO_RW_CHECKS

  Performs write/write checking only. If a write/write conflict is detected, then your transaction cannot commit; read/write conflicts are ignored. This mode allows an occasional out-of-date entry to overwrite a more current one. You can use object locks to enforce more stringent control if you can anticipate the problem.

The FULL_CHECKS setting is the most convenient and efficient if:

- you are not sharing data with other sessions, or
- you are reading data but not writing, or
- you are writing a limited amount of shared data and you can tolerate not being able to commit your work sometimes, or
- you are not likely to write the same objects as another concurrent session.

Concurrency Management

As the application designer, you determine your approach to concurrency control.

- Using the *optimistic* approach to concurrency control, you simply read and write objects as if you were the only user. The object server detects conflicts with other sessions only at the time you try to commit your transaction. Your chance of being in conflict with other users increases with the time between commits and the size of your read set and you write set.

  Although easy to implement in an application, this approach entails the risk that you might lose the work you've done if conflicts are detected and you are unable to commit.

- Using the pessimistic approach to concurrency control, you detect and prevent conflicts by explicitly requesting *locks* that signal your intentions to read or write objects. By locking an object, other users are unable to use the object in a way that conflicts with your purposes. If you are unable to acquire a lock, then someone else has already locked the object and you cannot use the object. You can then abort the transaction immediately instead of doing work that can't be committed.

- Using R*educed Conflict Classes* to perceive a write/write conflict and further test the changes to see if they can truly be added concurrently. In some cases, allowing operations to succeed leaves the object in a consistent state, even though a write conflict is detected.

  The GemStone reduced-conflict classes work well in situations that otherwise experience unnecessary conflicts. These classes are: RcCounter, RcIdentityBag, RcQueue, and RcKeyValueDictionary. See "Classes That Reduce the Chance of Conflict" on page 6-26.

Transaction Modes

You use GemStone in either of two modes:

- *automatic transaction mode*

  In this mode, GemStone begins a transaction when you log in, and starts a new one after each commit or abort message. In this default mode, you are in a transaction the entire time you are logged into a GemStone session. If the work you are doing requires committing to the repository frequently, you need to use the automatic transaction mode as you cannot make permanent changes to the repository when you are outside a transaction.

- *manual transaction mode*

  In manual transaction mode, you may be logged in and outside of a transaction. You explicitly control whether your session can commit. Although when you log in a transaction is started for you, you can set the transaction mode to manual, which aborts the current transaction and leaves you outside a transaction. Then you can start a transaction when you are ready to commit. Manual transaction mode provides a method of minimizing the transactions, while still managing the repository for concurrent access.

  In this state, you can view the repository, browse objects, and make computations based upon object values. You cannot make permanent any changes, nor add any new objects you may have created while outside a transaction. You can start a transaction at any time during a session, and carry temporary results you may have computed while outside a transaction into your new transaction. Here they can be committed, subject to the usual constraints of conflict-checking.

To determine the transaction mode you are in, print the results of sending the message:

```
System transactionMode
```

Changing Transaction Mode

To change to manual transaction mode, send the message:

```
System transactionMode: #manualBegin
```

This message aborts the current transaction and changes the transaction mode. It does not start a new transaction, but it does provide a fresh view of the repository. (Use `#autoBegin` to returns to automatic transaction mode.)

Beginning New Manual Transactions

In manual transaction mode, you can start a transaction by sending the message:

```
System beginTransaction
```

This message gives you a fresh view of the repository and starts a transaction. When you commit or abort this new transaction, you will again be outside of a transaction until you either explicitly begin a new one or change transaction modes.

If you send the message `System beginTransaction` while you are already in a transaction, GemStone does an abort.

You can determine whether you are currently in a transaction by sending the message:

```
System inTransaction
```

This message returns true if you are in a transaction and false if you are not.

Committing Transactions

Committing a transaction has two effects:

- It makes your new and changed objects visible to other users as a permanent part of the repository.
- It makes visible to you any new or modified objects that have been committed by other users in an up-to-date view of the repository.

When you tell GemStone to commit your transaction, the object server:

1. Checks whether other concurrent sessions have committed transactions that modify an object that you modified during your transaction.
2. Checks whether other concurrent sessions have committed transactions that modify an object that you have read during your transaction.
3. Checks for locks set by other sessions that indicate the intention to modify objects that you have read.

If none of these conditions is found, GemStone commits the transaction. The message `commitTransaction` commits the current transaction:

Example 6.1

```
UserGlobals at: #SharedDictionary put: SymbolDictionary new.

SharedDictionary at: #testData put: 'a string'.
      "modifies private view"
System commitTransaction.
      "commit the transaction, merging my private view
       of SharedDictionary with the committed repository"
%
```

The message `commitTransaction` returns true if GemStone commits your transaction and false if it can't. To find why your transaction failed to commit, you can send the message:

```
System transactionConflicts
```

This method returns a symbol dictionary in which the keys indicate the kind of conflict detected; they are one of a set of symbols as shown in Table 6.1.

Table 6.1 Transaction Conflict Keys

Key	Meaning
#'Read-Write'	ReadSet and WriteSetUnion conflict.
#'Write-Read'	WriteSet and ReadSetUnion conflict.
#'Write-Write'	WriteSet and WriteSetUnion conflict.
#'Read-ExclusiveLock'	ReadSet and ExclusiveLockSet conflict.
#'Write-ReadLock'	WriteSet and ExclusiveLockSet conflict.
#'Write-WriteLock'	WriteSet and WriteLockSet conflict.
#Rc-Write-Write	Write-Write conflict on an instance of a reduced conflict class.

If the transaction experienced no conflicts, an empty dictionary is returned. Conflict sets are cleared at the beginning of a commit and can be examined until the next commit.

Handling Commit Failure In A Transaction

If GemStone refuses to commit your transaction, the transaction read or wrote an object that another user modified and committed to the repository since your transaction began. Because you can't undo a read or a write operation, simply repeating the attempt to commit will not succeed.

You must abort the transaction in order to get a new view of the repository and, along with it, an empty read set and an empty write set. A subsequent attempt to run your code and commit the view can succeed. If the competition for shared data is heavy, subsequent transactions can also fail to commit. In this situation, locking objects that are frequently modified by other transactions gives you a better chance of committing.

One common cause of a write-write conflict occurs when two users simultaneously try to override the same inherited method, even though the two users are implementing their methods in two different subclasses. If both of the subclasses to which the users are adding the method have been committed, but neither subclass implemented the inherited method, the first user who tries to commit will succeed, but the second user will get a write-write conflict for that method in the superclass's method dictionary. In this case, the second user can commit after aborting the transaction, because the first user will have completed the implementation and will no longer be in the first user's write set.

Indexes and Concurrency Control

Building an index on a collection effectively writes all of the elements in the collection and all values of instance variables that participate in the index (those that appear in the indexed path). As a result, it is possible to experience conflict on objects that you have not explicitly written.

Consider the following expression:

```
myEmps createEqualityIndexOn: 'department.manager.name'
```

This expression writes all of the employees in myEmps, the department instance variable in every instance of Employee, the manager instance variable in every instance of Department, and the name instance variable in every instance of Manager (assuming such classes).

If one transaction builds an index on salary into a collection of employees and a second transaction modifies the salary of one of the employees in the collection, the two transactions will conflict on the modified employee. This conflict occurs even though the first transaction has not explicitly modified any of the employees, and the second transaction has not explicitly modified the collection.

Although unlikely, it is possible that you can encounter conflict on the internal indexing structures used by GemStone. For example, if two transactions modify the salaries of different employees that participate in the same indexed set, it is possible that both transactions will modify the same internal indexing structure and therefore conflict, despite the fact that neither transaction has explicitly accessed an object written by the other transaction.

To check this possibility, examine the dictionary returned by evaluating `System transactionConflicts`. If it refers to instances of the classes DependencyList, IndexList, or any class containing "Btree" in its name, it is likely that you have experienced a conflict on some portion of an indexing structure. In that case, you can abort the transaction and try the modification again.

Aborting Transactions

If GemStone refuses to commit your modifications, your view remains intact with all of the new and modified objects it contains. However, your view now also includes other users' modifications to objects that are visible to you, but that you have not modified. You must take some action to save the modifications in your session or in a file outside GemStone.

Then you need to *abort* the transaction. This discards all of the modifications from the aborted transaction, and gives you a new view containing the shared, committed objects. Depending on the activities of other users, you can repeat your operations using the new values and commit the new transaction without encountering conflicts.The message `abortTransaction` discards the modified objects in your view. If you are automatic transaction mode, this message also begins a new transaction.

Example 6.2

```
SharedDictionary at: #testData put: 'a string'.
      "modifies private view"

System abortTransaction.
      "discard the modified copy of SharedDictionary
       and all other modified objects, get a new view,
       and start a new transaction"
```

Aborting a transaction discards any changes you have made to shared objects during the transaction. However, work you have done locally within your own interface is not affected by an `abortTransaction`. GemStone gives you a new

view of the repository that does not include any changes you made to permanent objects during the aborted transaction. The new view includes changes committed by other users since your last transaction started. Objects that you have created locally in your own application, remain until you remove them or end your session.

Updating the View Without Committing or Aborting

The message `continueTransaction` gives you a new, up-to-date view of other users' committed work without discarding the objects you have modified in your current session.

The message `continueTransaction` returns true if your uncommitted changes do not conflict with the current state of the repository; it returns false is the repository has changed.

Unlike `commitTransaction` and `abortTransaction`, `continueTransaction` does not end your transaction. It has no effect on object locks. `continueTransaction` does not discard any changes you have made or commit any changes. Objects you have modified or created do not become visible to other users.

Work you have done locally within your own interface is not affected by a `continueTransaction`. Objects that you have created in your own application remain. Similarly, any execution that you have begun continues, unless the execution explicitly depends upon a successful commit operation.

Being Signaled to Abort

As mentioned earlier, being in a transaction incurs certain costs. When you are in a transaction, GemStone waits until you commit or abort to reclaim obsolete objects in your view. When you are outside of a transaction, GemStone warns y9u when your view is outdated, sending your session the error #rtErrSignalAbort. You are allowed a certain amount of time to abort your current view, as specified in the STN_GEM_ABORT_TIMEOUT parameter in your configuration file. (This parameter is described in the *GemStone System Administrator's Guide*.) When you abort your current view (by sending the message `System abortTransaction`), GemStone can reclaim storage and you get a fresh view of the repository.

If you do not respond within the specified time period, the object server forces an abort, sending your session the error #abortErrLostOtRoot. This indicates that your view of the repository has been recomputed, and copies of objects that your application had been holding may no longer be valid.

You lose nothing you cannot retrieve — work you have done locally (such as references to objects within your application) is retained, and you still cannot commit work to the repository when running outside of a transaction. However, you must read again those objects that you had previously read from the repository, and recompute the results of any computations performed on them, because the object server no longer guarantee that the application values are valid.

Your GemStone session controls whether it receives the error message #rtErrSignalAbort. To enable receiving it, send the message:

```
System enableSignaledAbortError
```

To disable receiving it, send the message:

```
System disableSignaledAbortError
```

To determine whether receiving this error message is presently enabled or disabled, send the message:

```
System signaledAbortErrorStatus
```

This method returns true if the error message is enabled, and false if it is disabled. By default, GemStone sessions disable receiving this error message. The GemBuilder interfaces may change this default. If you wish to be notified of the error (for an exception handler for the circumstance), then you must explicitly enable the signaled abort error.

NOTE:
When running outside a transaction, GemStone behaves as described, possibly aborting your view, whether or not your session receives the error message for a signaled abort.The GemBuilder interfaces can handle this abort request for you. See your GemBuilder manual for details.

6.3 Controlling Concurrent Access With Locks

If many users are competing for shared data in your application, or you can't tolerate even an occasional inability to commit, then you can implement pessimistic concurrency control by using locks.

Locking an object is a way of telling GemStone (and, indirectly, other users) your intention to read or write the object. Holding locks prevents transactions whose activities would conflict with your own from committing changes to the repository. Unless you specify otherwise, GemStone locks persist across aborts. If you lock on an object and then abort, your session still holds the lock after the abort. Aborting the current transaction (and starting another, if you are in manual

transaction mode) gives you an up-to-date value for the locked object without removing the lock.

Remember, locking improves one user's chances of committing only at the expense of other users. Use locks sparingly to prevent an overall degradation of system performance.

Locking and Manual Transaction Mode

GemStone permits you to request any kind of lock, no matter your transaction mode or whether you are in a transaction. When you are using manual transaction mode and running outside of a transaction, however, you are not allowed to commit the results of your operations. Requesting a lock under such circumstances is not helpful, and can adversely affect other users' ability to get work done. It may be useful to request a lock to determine whether an object is dirty, and therefore to ascertain whether your view of it is current and valid. Otherwise, do not recommended to request a lock when outside a transaction.

Lock Types

GemStone provides three kinds of locks: *read*, *write*, and *exclusive*. A session may hold only one kind of lock on an object at a time.

Read Locks

Holding a read lock on an object means that you can use the object's value, and then commit without fear that some other transaction has committed a new value for that object during your transaction. Another way of saying this is that holding a read lock on an object guarantees that other sessions cannot:

- acquire a write or exclusive lock on the object, or
- commit if they have written the object.

To understand the utility of read locks, imagine that you need to compute the average age of a large number of employees. While you are reading the employees and computing the average, another user changes an employee's age and commits (in the aftermath of the birthday party). You have now performed the computation using out-of-date information. You can prevent this frustration by read-locking the employees at the outset of your transaction; this prevents changes to those objects.

Multiple sessions can hold read locks on the same object. A maximum of 1 million read locks can be held concurrently, but a maximum of 2000 is recommended.

NOTE:
If you have a read lock on an object and you try to write that object, your attempt to commit that transaction will fail.

Write Locks

Holding a write lock on an object guarantees that you can write the object and commit. That is, it ensures that you won't find that someone else has prevented you from committing by writing the object and committing it before you, while your transaction was in progress. Another way of looking at this is that holding a write lock on an object guarantees that other sessions cannot:

- acquire any kind of lock on the object, or
- commit if they have written the object.

Write locks are useful, for example, if you want to change the addresses of a number of employees. If you write-lock the employees at the outset of your transaction, you prevent other sessions from modifying one of the employees and committing before you can finish your work. This guarantees your ability to commit the changes.

Write locks differ from read locks in that only one session can hold a write lock on an object. In fact, if a session holds a write lock on an object, then no other session can hold any kind of lock on the object. This prevents another session from receiving the assurance implied by a read lock: that the value of the object it sees in its view will not be out of date when it attempts to commit a transaction.

Exclusive Locks

An exclusive lock is like a write lock in that it guarantees your ability to write an object. It goes beyond a write lock by guaranteeing that other sessions cannot:

- acquire any kind of lock on the object, or
- commit if they have written or read the object.

Exclusive locks are most easily understood by considering an example.

Suppose once more that you want to change the addresses of a set of employees. Simply write-locking the employees prevents other users from changing the employees before you have finished with them. It does not, however, prevent another user from reading the employees' addresses (to determine geographic distribution, perhaps), and then writing that derived information in other objects.

To prevent another user from obtaining possibly inaccurate data, you must acquire an exclusive lock on every employee.

GemStone's exclusive locks correspond to what traditional data management systems call exclusive locks, or sometimes just write locks. By contrast, GemStone's write locks are not exclusive in the conventional sense, because other sessions can read a write-locked object optimistically (that is, without holding a lock) and still commit.

Acquiring Locks

The kernel class System is the receiver of all lock requests. The following statements request one lock of each kind:

Example 6.3

```
System readLock: SharedDictionary.
System writeLock: myEmployees.
System exclusiveLock: someMoreData.
```

When locks are granted, these messages return System.

Commits and aborts do not necessarily release locks, although locks can be set up so that they will do so. Unless you specify otherwise, once you acquire a lock, it remains in place until you log out or remove it explicitly. (Subsequent sections explain how to remove locks.) Variants of these messages for locking collections of objects en masse are described in the section "Locking Collections Of Objects Efficiently" on page 6-19.

When a lock is requested, GemStone grants it unless one of the following conditions is true:

- You do not have suitable authorization. Read locks require read authorization; write locks and exclusive locks require write authorization.
- The object is an instance of SmallInteger, Boolean, Character, or nil. Trying to lock these special, atomic objects is meaningless.
- The object is already locked in an incompatible way by another session (remember, only read locks can be shared).

Lock Denial

If you request a lock on an object and another session already holds a conflicting lock on it, then GemStone denies your request; GemStone does not automatically wait for locks to become available.

If you use one of the simpler lock request messages (such as `readLock:`), lock denial generates an error. If you want to take some automatic action in response to the denial, use a more complex lock request message, such as the one described above. A lock denial causes GemStone to execute the block argument to `ifDenied:`. The method in Example 6.4 uses this technique to request a lock repeatedly until the lock becomes available:

Example 6.4

```
anObject := Object new.
%
Object subclass: #Dummy
   instVarNames: #()
   inDictionary: UserGlobals
%
method: Dummy
getReadLockOn: anObject
  "This method tries to lock anObject. If the lock is
   denied, it determines the kind of lock and the user who
   has locked the object."
System readLock: anObject
       ifDenied: [ ^ #[ System lockKind: anObject,
                        System lockOwners: anObject] ]
       ifChanged: [System abortTransaction].
%
Dummy new getReadLockOn: anObject
%
method: Dummy
getReadLockOn: anObject
System readLock: anObject
       ifDenied: [self getReadLockOn: anObject]
       ifChanged: [System abortTransaction]
%

Dummy new getReadLockOn: Object new
```

Dead Locks

You may never acquire a lock, no matter how long you wait. Furthermore, because GemStone does not automatically wait for locks, it does not attempt deadlock detection. It is your responsibility to limit the attempts to acquire locks in some way. For example, you can write a portion of your application in such a way that there is an absolute time limit on attempts to acquire a lock. Or you can let users know when locks are being awaited and allow them to interrupt the process if needed.

Dirty Locks

If another user has written an object and committed the change since your transaction began, then the value of the object in your view is out of date. Although you may be able to acquire a lock on the object, it is a *dirty lock* because you cannot use the object and commit, despite holding the lock.

This condition is trapped by the argument to the `ifChanged:` keyword following read lock request message:

```
System readLock: anObject ifDenied: [block1]
                          ifChanged: [block2].
```

Like its simpler counterpart, this message returns System if it acquires a lock on *anObject* without complications. It generates an error if the user has no authorization for acquiring the lock, or selects one of the blocks passed as arguments and executes that block, returning the block's value.

For example, if a conflicting lock is held on *anObject*, this message executes the block given as an argument to the keyword `ifDenied:`. Similarly, if *anObject* has been changed by another session, it executes the argument to `ifChanged:`. The following sections provide some suggestions about the code such blocks might contain.

For example:

Example 6.5

```
expectvalue %Object
run
anObject := Object new.
%
System readLock: anObject
       ifDenied: []
       ifChanged: [System abortTransaction]
%
```

To minimize your chances of getting dirty locks, lock the objects you need as early in your transaction as possible. If you encounter a dirty lock in the process, you can keep track of the fact and continue locking. After you finish locking, you can abort your transaction to get current values for all of the objects whose locks are dirty. For example:

Example 6.6

```
anObject := Object new.
%

| dirtyBag |
dirtyBag := IdentityBag new.
myEmployees do: [:anEmp |
   System readLock: anEmp
      ifDenied: []
      ifChanged: [ dirtyBag add: anEmp ] ].
dirtyBag isEmpty
   ifTrue: [ ^true ]
   ifFalse: [ System abortTransaction ].
%
```

Your new transaction can then proceed with clean locks.

Locking Collections Of Objects Efficiently

In addition to the locking request messages for single objects, GemStone provides locking request messages that can lock entire collections of objects. If the objects

you need to lock are already in collections, or if they can be gathered into collections without too much work, it is more efficient for you to use the collection-locking methods than to lock the objects individually.

The following statements request locks on each of the elements of three different collections:

Example 6.7

```
UserGlobals at: #myArray put: Array new;
   at: #myBag put: IdentityBag new;
   at: #mySet put: IdentitySet new.
%

System readLockAll: myArray.
System writeLockAll: myBag.
System exclusiveLockAll: mySet.
%
```

The messages used in this example are similar to the simple, single-object locking-request messages (such as `readLock:`) that you've already seen. If a clean lock is acquired on each element of the argument, these messages return System. If you lack the proper authorization for any object in the argument, GemStone generates an error and grants no locks.

The difference between these methods and their single-object counterparts is in the handling of other errors. The system does not immediately halt to report an error if an object in the collection is changed, or if a lock must be denied because another session has already locked the object. Instead, the system continues to request locks on the remaining elements, acquiring as many locks as possible. When the method finishes processing the entire collection, it generates an error. In the meantime, however, all locks that you acquired remain in place.

You might want to handle these errors from within your Smalltalk program instead of letting execution halt. For this purpose, class System provides collection-locking methods that pass information about unsuccessful lock requests to blocks that you supply as arguments. For example:

```
System writeLockAll: aCollection ifIncomplete: aBlock
```

The argument *aBlock* that you supply to this method must take three arguments. If locks are not granted on all elements of *aCollection* (for any reason except authorization failure), the method passes three arrays to *aBlock* and then executes the block. The first array contains all elements of *aCollection* for which locks were

denied; the second contains all elements for which dirty locks were granted; and the third is empty (it is there for compatibility with previous versions of GemStone).

You can then take appropriate actions within the block. For example:

Example 6.8

```
classmethod: Dummy
handleDenialOn: deniedObjs
^ deniedObjs
%
classmethod: Dummy
getWriteLocksOn: aCollection
System writeLockAll: aCollection
   ifIncomplete: [:denied :dirty :unused |
      denied isEmpty ifFalse: [self handleDenialOn: denied].
      dirty isEmpty ifFalse: [System abortTransaction] ]
%

System readLockAll: myEmployees
%

Dummy getWriteLocksOn: myEmployees
%
```

Upgrading Locks

On occasion, you might want to *upgrade* a lock; that is, change a read lock to a write lock or a write lock to an exclusive lock. You might initially intend to read an object, only to discover later that you must also write the object. Although you could take the risk of writing the object optimistically (this is, without a write lock), you might wish to ensure your ability to commit by first upgrading the lock to a write lock.

GemStone currently provides no built-in support for upgrading locks. However, you can remove the lock you currently hold and then immediately request an upgraded lock. In the situation described in the previous paragraph, for example, you could remove the read lock on the object and then immediately request a write lock.

It is important to request the upgraded lock immediately, because between the time that the lock is removed, and the time that the upgraded lock is requested, another session has the opportunity to lock the object, or to write it and commit.

Locking and Indexed Collections

When indexes are present, locking can fail to prevent conflict. The reasons are similar to those presented above in the section "Indexes and Concurrency Control" on page 6-10. Briefly, GemStone maintains indexing structures in your view and does not lock these structures when an indexed collection or one of its elements is locked. Therefore, despite having locked all of the visible objects that you touched, you can be unable to commit.

Specifically, this means that:

- *if* an object is either an element of an indexed collection, or participates in an index (meaning it is a component of an element bearing an index);
- *and* another session can access the object, an indexed collection of which the object is a member, or one of its predecessors along the same indexed path—

  then locking the object does not guarantee that you can commit after reading or writing the object.

Therefore, don't rely on locking an object if the object participates in an index.

Removing or Releasing Locks

Once you lock an object, its default behavior is to remain locked until you either log out or explicitly remove the lock; unless you specify otherwise, locks persist through aborts and commits. In general, remove a lock on an object when you have used the object, committed the resulting values to the repository, and no longer foresee an immediate need to maintain control of the object.

Class System provides three lock removal messages: one for removing a lock from a single object, one for removing locks from each of a collection of objects, and one for removing all locks held by your session. The following statement removes any lock you might hold on anObject:

```
System removeLock: anObject
```

If anObject is not locked, GemStone does nothing. If another session holds a lock on anObject, this message has no effect on the other session's lock.

The following statement removes any locks you might hold on the elements of aCollection.

```
System removeLockAll: aCollection.
```

If you intend to continue your session, but the next transaction is to work on a different set of objects, you might wish to remove all the locks held by your session. The following statement attempts to commit the present transaction and removes all locks it holds (even if the commit did not succeed):

```
System commitTransaction; removeLocksForSession
```

If you wish to commit your transaction and release all the locks you hold in one operation, the following statement is optimized to do so as efficiently as possible:

```
System commitAndReleaseLocks
```

If your transaction fails to commit, all locks are held instead of released.

Releasing Locks Upon Aborting or Committing

After you have locked an object, you can add it to either of two special sets. One set contains the locked objects whose locks you wish to release as soon as you commit your current transaction. The other set contains the locked objects whose locks you wish to release as soon as you either commit or abort your current transaction. Executing `continueTransaction` does not release the locks in either set.

The following statement adds a locked object to the set of objects whose locks are to be released upon the next commit:

```
System addToCommitReleaseLocksSet: aLockedObject
```

The following statement adds a locked object to the set of objects whose locks are to be released upon the next commit or abort:

```
System addToCommitOrAbortReleaseLocksSet: aLockedObject
```

The following statement adds the locked elements of a collection to the set of objects whose locks are to be released upon the next commit:

```
System addAllToCommitReleaseLocksSet: aLockedCollection
```

The following statement adds the locked elements of a collection to the set of objects whose locks are to be released upon the next commit or abort:

```
System addAllToCommitOrAbortReleaseLocksSet: aLockedCollection
```

NOTE

If you add an object to one of these sets and then request an updated lock on it, the object is removed from the set.

You can remove objects from these sets without removing the lock on the object. The following statement removes a locked object from the set of objects whose locks are to be released upon the next commit:

```
System removeFromCommitReleaseLocksSet: aLockedObject
```

The following statement removes a locked object from the set of objects whose locks are to be released upon the next commit or abort:

```
System removeFromCommitOrAbortReleaseLocksSet: aLockedObject
```

The following statement removes the locked elements of a collection from the set of objects whose locks are to be released upon the next commit:

```
System removeAllFromCommitReleaseLocksSet: aLockedCollection
```

The following statement removes the locked elements of a collection from the set of objects whose locks are to be released upon the next commit or abort:

```
System removeAllFromCommitOrAbortReleaseLocksSet: aLockedCollection
```

You can also remove all objects from either of these sets with one message. The following statement removes all objects from the set of objects whose locks are to be released upon the next commit:

```
System clearCommitReleaseLocksSet
```

The following statement removes all objects from the set of objects whose locks are to be released upon the next commit or abort:

```
System clearCommitOrAbortReleaseLocksSet
```

The statement `System commitAndReleaseLocks` also clears both sets if the transaction was successfully committed.

Inquiring About Locks

GemStone provides messages for inquiring about locks held by your session and other sessions. Most of these messages are intended for use by the data curator, but several can be useful to ordinary applications.

The message `sessionLocks` gives you a complete list of all the locks held by your session. This message returns a three-element array. The first element is an array

of read-locked objects; the second is an array of write-locked objects; the third is an array of exclusively locked objects.

The following code uses this information to remove all write locks held by the current session:

```
System removeLockAll: (System sessionLocks at: 2)
```

Another useful lock inquiry message is `systemLocks`, which reports locks on all objects held by all sessions currently logged in to the repository. The only exception is that `systemLocks` does not report on any locks other sessions are holding on their temporary objects—that is, objects that they have never committed to the repository. Because such objects are not visible to you in any case, this omission is not likely to cause a problem. The message `systemLocks` can help you discover the cause of a conflict.

Another lock inquiry message, `lockOwners:` *anObject*, is useful if you've been unable to acquire a lock because of conflict with another session. This message returns an array of SmallIntegers representing the sessions that hold locks on *anObject*. The method in the following example uses `lockOwners:` to build an array of the userIDs of all users whose sessions hold locks on a particular object.

Example 6.9

```
classmethod: Dummy
getNamesOfLockOwnersFor: anObject
| userIDArray sessionArray |
sessionArray := System lockOwners: anObject.
userIDArray := Array new.
sessionArray do:
      [:aSessNum | userIDArray add:
                (System userProfileForSession: aSessNum) userId].
^userIDArray
%

Dummy getNamesOfLockOwnersFor: (myEmployees detect: {:e | e.name =
'Conan' })
%
```

You can test to see whether an object is included in either of the sets of locked objects whose locks are to be released upon the next abort or commit operation.

The following statement returns true if the object provided as an argument is included in the set of objects whose locks are to be released upon the next commit:

```
System commitReleaseLocksSetIncludes: anObject
```

The following statement returns true if the object provided as an argument is included in the set of objects whose locks are to be released upon the next commit or abort:

```
System  commitOrAbortReleaseLocksSetIncludes: anObject
```

For information about the other lock inquiry messages, see the description of class System in the *GemStone Kernel Reference*.

6.4 Classes That Reduce the Chance of Conflict

Often, concurrent access to an object is structural, but not semantic. GemStone detects a conflict when two users access the same object, even when respective changes to the objects do not collide. For example, when two users both try to add something to a bag they share, GemStone perceives a write/write conflict on the second add operation, although there is really no reason why the two users cannot both add their objects. As human beings, we can see that allowing both operations to succeed leaves the bag in a consistent state, even though both operations modify the bag.

A situation such as this can cause spurious conflicts. Therefore, GemStone provides four reduced-conflict classes that you can use instead of their regular counterparts in applications that might otherwise experience too many unnecessary conflicts. These classes are:

- RcCounter,
- RcIdentityBag,
- RcQueue, and
- RcKeyValueDictionary.

Using these classes allows a greater number of transactions to commit successfully, improving system performance. However, in order to determine whether it is appropriate for your application to use these reduced-conflict classes, you need to be aware of the costs:

- The reduced-conflict classes use more storage than their ordinary counterparts.

- When using instances of these classes, at times your application may take longer to commit transactions.
- Under certain circumstances, instances of these classes can hide conflicts from you that you indeed need to know about. They are not always appropriate.
- These classes are not exact copies of their regular counterparts. In certain cases they may behave slightly differently.

"Reduced conflict" does not mean "no conflict." The reduced-conflict classes do not circumvent normal conflict mechanisms; under certain circumstances, you will still be unable to commit a transaction. These classes use different implementations or more sophisticated conflict-checking code to allow certain operations that human analysis has determined need not conflict. They do not allow *all* operations. Using these classes eliminates read/write conflicts on their instances, and significantly reduces write/write conflicts.

NOTE:
Unlike other Dictionaries, the class RcKeyValueDictionary does not support indexing because of its position in the class hierarchy.

RcCounter

The class RcCounter can be used instead of a simple number in order to keep track of the amount of something. It allows multiple users to increment or decrement the amount at the same time without experiencing conflicts.

The class RcCounter is not a kind of number. It encapsulates a number—the counter—but it also incorporates other intelligence; you cannot use an RcCounter to replace a number anywhere in your application. It only increments and decrements a counter.

For example, imagine an application to keep track of the number of items in a warehouse bin. Workers increment the counter when they add items to the bin, and decrement the counter when they remove items to be shipped. This warehouse is a busy place; if each concurrent increment or decrement operation produces a conflict, work slows unacceptably.

Furthermore, the conflicts are mostly unnecessary. Most of the workers can tolerate a certain amount of inaccuracy in their views of the bin count at any time. They do not need to know the exact number of items in the bin at every moment; they may not even worry if the bin count goes slightly negative from time to time. They may simply trust that their views are not completely up-to-date, and that their fellow workers have added to the bin in the time since their views were last refreshed. For such an application, an RcCounter is helpful.

Instances of RcCounter understand the messages `increment` (which increments by 1), `decrement` (which decrements by 1), and `value` (which returns the number of elements in the counter). Additional protocol allows you to increment or decrement by specified numbers, to decrement unless that operation would cause the value of the counter to become negative, in which case an alternative block of code is executed instead, or to decrement unless that operation would cause the value of the counter to be less than a specified number, in which case an alternative block of code is executed instead.

For example, the following operations can all take place concurrently from different sessions without causing a conflict:

Example 6.10

```
!session 1
UserGlobals at: #binCount put: RcCounter new.
System commitTransaction.
%
!session 2
binCount incrementBy: 48.
System commitTransaction.
%
!session 1
binCount incrementBy: 24.
System commitTransaction.
%
!session 3
binCount decrementBy: 144
   ifLessThan: -24
   thenExecute: [^'Not enough widgets to ship today.'].
System commitTransaction.
%
```

RcCounter is not appropriate for all applications—for example, it would not be appropriate to use in an application that keeps track of the amount of money in a shared checking account. If two users of the checking account both tried to withdraw more than half of the balance at the same time, an RcCounter would allow both operations without conflict. Sometimes, however, you need to be warned—for example, of an impending overdraft.

RcIdentityBag

The class RcIdentityBag provides much of the same functionality as IdentityBag, including the expected behavior for `add:`, `remove:`, and related messages. However, no conflict occurs on instances of RcIdentityBag when:

- any number of users read objects in the bag at the same time;
- any number of users add objects to the bag at the same time;
- one user removes an object from the bag while any number of users are adding objects; and
- any number of users remove objects from the bag at the same time, as long as no more than one of them tries to remove the last occurrence of an object.

When your session and others remove different occurrences of the same object, you may sometimes notice that it takes a bit longer to commit your transaction.

The class RcIdentityBag does not implement the entire set of methods implemented in the class IdentityBag. For example, IdentityBag implements a suite of methods to perform set arithmetic, which are not implemented in the class RcIdentityBag. If you wish to use any of these methods, you can copy your instance of RcIdentityBag as an instance of IdentityBag using the message `asIdentityBag`. You can then perform the operation on the resulting instance of IdentityBag and store the result.

Finally, indexing an instance of RcIdentityBag does diminish somewhat its "reduced-conflict" nature, because of the possibility of a conflict on the underlying indexing structure. (See "Indexes and Concurrency Control" on page 6-10 for a more complete explanation of this possibility.) However, even an indexed instance of RcIdentityBag reduces the possibility of a transaction conflict, compared to an instance of IdentityBag, indexed or not.

RcQueue

The class RcQueue approximates the functionality of a first-in-first-out queue, including the expected behavior for `add:`, `remove:`, `size`, and `do:`, which evaluates the block provided as an argument for each of the elements of the queue. No conflict occurs on instances of RcQueue when:

- any number of users read objects in the queue at the same time;
- any number of users add objects to the queue at the same time;
- one user removes an object from the queue while any number of users are adding objects.

If more than one user removes objects from the queue, they are likely to experience a write/write conflict. When a commit fails for this reason, the user loses all changes made to the queue during the current transaction, and the queue remains in the state left by the earlier user who made the conflicting changes.

RcQueue approximates a first-in-first-out queue, but it cannot implement such functionality exactly because of the nature of repository views during transactions. The consumer removing objects from the queue sees the view that was current when his or her transaction began. Depending upon when other users have committed their transactions, the consumer may view objects added to the queue in a slightly different order than the order viewed by those users who have added to the queue. For example, suppose one user adds object A at 10:20, but waits to commit until 10:50. Meanwhile, another user adds object B at 10:35 and commits immediately. A third user viewing the queue at 10:30 will see neither object A nor B. At 10:35, object B will become visible to the third user. At 10:50, object A will also become visible to the third user, and will furthermore appear earlier in the queue, because it was created first.

Objects removed from the queue always come out in the order viewed by the consumer.

Because of the manner in which RcQueues are implemented, reclaiming the storage of objects that have been removed from the queue actually occurs when new objects are added. If a session adds a great many objects to the queue all at once and then does not add any more as other sessions consume the objects, performance can become degraded, particularly from the consumer's point of view. In order to avoid this, the producer can send the message `cleanupMySession` occasionally to the instance of the queue from which the objects are being removed. This causes storage to be reclaimed from obsolete objects.

NOTE:
If you subclass and reimplement these methods, build in a check for nils. Because of lazy initialization, the expected subcomponents of the RcQueue may not exist yet.

RcKeyValueDictionary

The class RcKeyValueDictionary provides the same functionality as KeyValueDictionary, including the expected behavior for `at:`, `at:put:`, and `removeKey:`. However, no conflict occurs on instances of RcKeyValueDictionary when:

- any number of users read values in the dictionary at the same time;
- any number of users add keys and values to the dictionary at the same time, unless a user tries to add a key that already exists;
- any number of users remove keys from the dictionary at the same time, unless more than one user tries to remove the same key at the same time; and
- any number of users perform any combination of these operations.

Chapter 7

Object Security and Authorization

This chapter explains how to set up the object security required for developing an application and for running the completed application. It covers:

How GemStone Security Works
describes the Gemstone object security model.

An Application Example and A Development Example
provides two examples for defining and implementing object security for your projects.

Assigning Objects to Segments
summarizes the messages for reporting your current segment, changing your current segment, and assigning a segment to simple and complex objects.

Privileged Protocol for Class Segment
defines the system privileges for creating or changing segment authorization.

Segment-related Methods
lists the methods that affect segments.

7.1 How GemStone Security Works

GemStone provides security at several levels:

- Login authorization keeps unauthorized users from gaining access to the repository;
- Privileges limit ability to execute special methods affecting the basic functioning of the system (for example, the methods that reclaim storage space); and
- Object level security allows specific groups of users access to individual objects in the repository.

Login Authorization

You log into GemStone through any of the interfaces provided: the GemStone Smalltalk Interface, Topaz, the C++ interface, or the C interface (see the appropriate interface manual for details). Whichever interface you use, GemStone requires the presentation of a *user ID* (a name or some other identifying string) and a password. If the user ID and password pair match the user ID and password pair of someone authorized to use the system, GemStone permits interaction to proceed; if not, GemStone severs the logical connection.

The GemStone system administrator, or someone with equivalent privileges (see below), establishes your user ID and password when he or she creates your *UserProfile*. The GemStone system administrator can also configure a GemStone system to monitor failures to log in, and to note the attempts in the Stone log file after a certain number of failures have occurred within a specified period of time. A system can also be configured to disable a user account after a certain number of failed attempts to log into the system through that account. See the *GemStone System Administration Guide* for details.

The UserProfile

Each instance of UserProfile is created by the system administrator. The UserProfile is stored with a set of all other UserProfiles in a set called AllUsers. The UserProfile contains:

- Your UserID and Password.
- A SymbolList (the list of symbols (objects) that the user has access to -- UserGlobal and Globals) for resolving symbols at compile-time. Chapter 6, “Symbol Resolution and Object Sharing,” discusses these topics, so they are not talked about in this chapter.
- The groups to which you belong and any special system privileges you may have.
- A defaultSegment to assign your session at login.
- Your language and character set used for internationalization.

See the *GemStone System Administration Guide* for instructions about creating UserProfiles.

System Privileges

Actions that affect the entire GemStone system are tightly controlled by *privileges* to use methods or access instances of the System, UserProfile, Segment, and Repository classes. Privileges are given to individual UserProfile accounts to access various parts of GemStone or perform important functions such as storage reclamation.

The privileged messages for the System, UserProfile, Segment and Repository Classes are described in the *GemStone Kernel Reference Guide*, and their use is discussed in the *GemStone System Administration Guide*.

Object Level Security

GemStone Object Level Security allows you to:

- abstractly group objects;
- specify who owns the objects;
- specify who can read them; and
- specify who can write them.

Each site designs a custom scheme for its data security. Objects can be secured for selective read or write access by a group or individual users. All objects in the GemStone system are subject to this object level security, just as all access goes through the authorization system.

The GemStone Segment class facilitates this security.

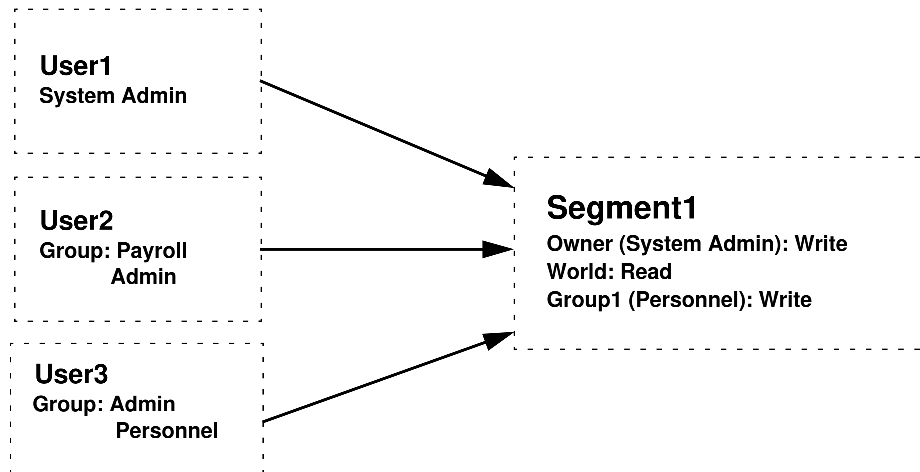
Segments

S*egments* are tags that identify the access various users have to the objects tagged by the segment. All objects assigned to a segment also have exactly the same protection; that is, if you can read or write one object assigned to a certain segment, you can read or write them all. Each segment is owned by a single user, and all objects assigned to the same segment have the same owner. Groups of users can have read, write, or no access to a segment. Likewise, any authorized GemStone user can have read, write, or no access to a segment.

Whenever an application tries to access an object, GemStone compares the object's authorization attributes in the segment associated with the object with those of the user whose application is attempting access. If the user is appropriately authorized, the operation proceeds. If not, GemStone returns an error notification

The user name, group membership, and segment authorization control access to objects, as shown by Figure 7.6:

Figure 7.1 User Access to Application Segment1



Three users access this application:

- The **System Administrator** owns segment1 and can read and write the objects assigned to it.
- **User3** belongs to the Personnel group, which authorizes read and write access to Segment1's objects.
- **User2** doesn't belong to a group that can access Segment1, but can still read those objects, because Segment1 gives read authorization to all GemStone users.

Because segments are objects, access to a segment object is controlled by the segment it is assigned to, exactly like access to any other object. Segment objects are usually assigned to the DataCurator segment. The access information stored in the segment object's own *authorizations* instance variable, which controls access to the objects assigned to that segment, does not control access to the segment object itself.

Objects do not “belong” to a segment. It is more correct to say that objects are associated with a segment. Although objects know which segment they are assigned to, segments do not know which objects are assigned to them. Segments are not meant to organize objects for easy listing and retrieval. For those purposes, you must turn to symbol lists, which are described in Chapter 3, “Name Resolution and Object Sharing“.

Default Segment and Current Segment

You are assigned a *default* segment as part of your UserProfile. When you login to GemStone, your Session uses this default segment as your current segment. Any objects you create are assigned to your current segment.

Class UserProfile has the message `defaultSegment`, which returns your default Segment. Sending the message `currentSegment:` to System changes your current segment:

Example 7.1

```
| aSegment mySegment |
mySegment := System myUserProfile defaultSegment.
aSegment := Segment newInRepository: SystemRepository.
"change my current segment to aSegment"
System currentSegment: aSegment
```

If you commit after changing segments, the new segment remains your current segment until you change the segment or log out. If you abort after changing your current segment, your current segment is reset from the UserProfile’s default segment. If the segment has already been committed, the abort operation has no effect on segment assignment.

Unnamed segments are often stored in a UserProfile, but named segments are stored in symbol dictionaries like other named objects. Private segments are typically kept in a user’s UserGlobals dictionary; segments for groups of users are typically kept in a shared dictionary.

You can also put segments in application dictionaries that appear only in the symbol lists of that application's users.

Example 7.2

```
| mySegment |
"get default Segment"
mySegment := System myUserProfile defaultSegment.
"compare with current Seg"
mySegment = System currentSegment

true
```

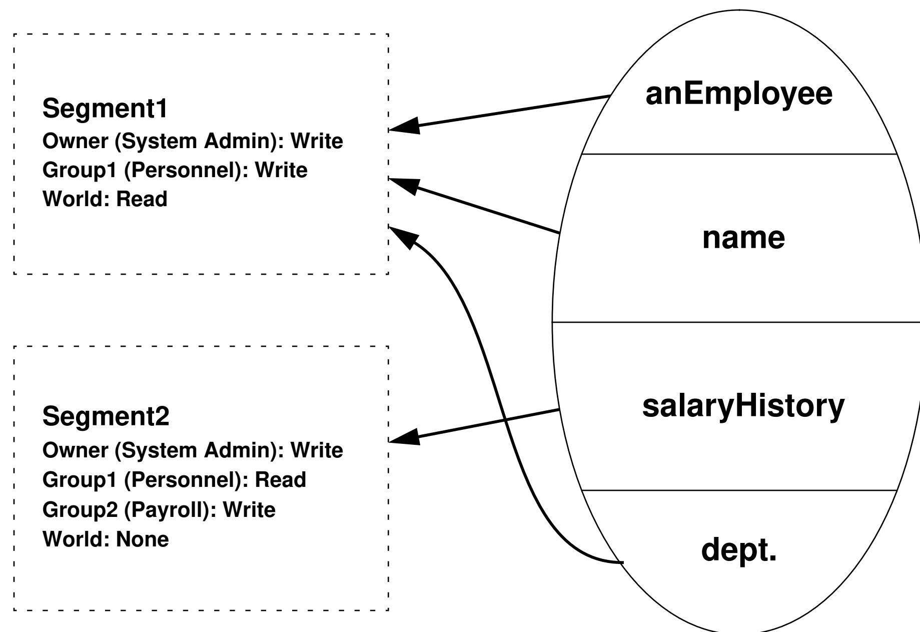
Objects and Segments

GemStone Object Security is defined for objects, not just instance variable slots. Your security scheme must be defined to protect sensitive data in separate objects, either by itself or as a member object of a customer class. Since each object has separate authorization, each object must be assigned separately.

Compound Objects

Usually, the objects you are working with are compound, and each part is an object in its own right, with its own segment assignment. For example, look at anEmployee in Figure 7.2. The contents of its instance variables (name, salary, and department) are separate objects that can be assigned to different segments. Salary is assigned to Segment2, which enforces more restricted access than Segment1.

Figure 7.2 Multiple Segment Assignments for a Compound Object



Every GemStone object is associated with a segment, except for objects of classes True, False, and SmallInteger. When objects are created, they are assigned to a default (the creator's *current*) segment unless specified otherwise.

Collections

When you assign collections of objects to segments, you must distinguish the container from the items it contains. Each of the items must also be assigned to the proper segment. Distinguishing between a collection and the objects it contains allows you to create collections most elements of which are publicly accessible, while some elements are sensitive.

Read and Write Authorization and Segments

Segments store authorization information that defines what a particular user or group member can do to the objects assigned to that segment. Three levels of authorization are provided:

write — means that a user can read and modify any of the segment's objects and create new objects associated with the segment.

read — means that a user can read any of the segment's objects, but cannot modify (write) them or add new ones.

none — means that a user can neither read nor write any of the segment's objects.

By assigning an object to a segment, you give the object the access information associated with that segment. Thus, all objects assigned to a segment have exactly the same protection; that is, if you can read or write one object assigned to a certain segment, you can read or write them all.

Controlling authorizations at the segment level rather than storing the information in each object makes them easy to change. Instead of modifying a number of objects individually, you just modify one segment object. This also keeps the repository smaller, eliminating the need for duplicate information in each of the objects.

How GemStone Responds to Unauthorized Access

GemStone immediately detects an attempt to read or write without authorization and responds by stopping the current method and issuing an error. When you successfully commit your transaction, GemStone verifies that you are still authorized to write in your current segment. If you are no longer authorized to do so, GemStone issues an error, and your default segment once again becomes your current segment. If you are no longer authorized to write in your default segment, GemStone terminates your session, and you are unable to log back in to GemStone. If this happens, see your system administrator for assistance.

Owner Authorization

The user that owns the segment controls what access other users have to it. The owner authorizes access separately for:

- a segment's *owner*
- *groups* of users (by name)
- the *world* of all GemStone users

These categories can overlap.

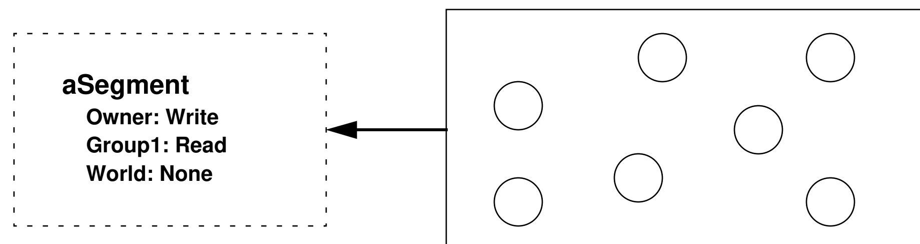
Whenever a program tries to read or write an object, GemStone compares the object's authorization attributes with those of the user who is attempting to do the reading or writing. If the user has authorization to perform the operation, it proceeds. If not, GemStone returns an error notification.

Groups

Groups are an efficient way to ensure that a number of GemStone users all will share the same level of access to objects in the repository, and all will be able to manipulate certain objects in the same ways.

Groups are typically organized as categories of users who have common interests or needs. In Figure 7.3, for example, Group1 was set up to allow a few users to read the objects in aSegment, while GemStone users in general aren't allowed any access.

Figure 7.3 User Access to a Segment's Objects



Membership in a group is granted by having the group name in one's UserProfile, and a group consists of all users with the group name in their profiles.

World Authorization

In addition to storing authorization for its owner and for some groups, a segment can also be told to authorize or to deny access by all GemStone users (*the world*.)

The message in class Segment that returns the rights of all users is `worldAuthorization`.

Changing the Authorization for World

A corresponding message, `worldAuthorization:` *anAuthSymbol*, sets the authorization for all GemStone users:

```
mySeg worldAuthorization: #read
```

Because of the way authorizations combine, changing access rights for the world may not alter a particular user's rights to a segment.

Segments in the Repository

The initial GemStone repository has three segments:

1. **DataCuratorSegment**

   This segment is defined in the Globals dictionary, and is owned by the DataCurator. All GemStone users, represented by world access, are authorized to read, but not write, objects associated with this segment. No groups are initially authorized to read or write in this segment.

   Objects in the DataCuratorSegment include the Globals dictionary, the SystemRepository object, all Segment objects, AllUsers (the set of all GemStone UserProfiles), AllGroups (the collection of groups authorized to read and write objects in GemStone segments), and each UserProfile object.

   NOTE:
   When GemStone is installed, only the DataCurator is authorized to write in this segment. To protect the objects in the DataCurator Segment against unauthorized modification, other users should not write in this segment.

2. **SystemSegment**

   This segment is defined in the Globals dictionary, and is owned by the SystemUser (who has write authorization for any of the objects in this segment). The world access is set to read, but not write, the objects in this segment. In addition, the group #System is authorized to write in this segment.

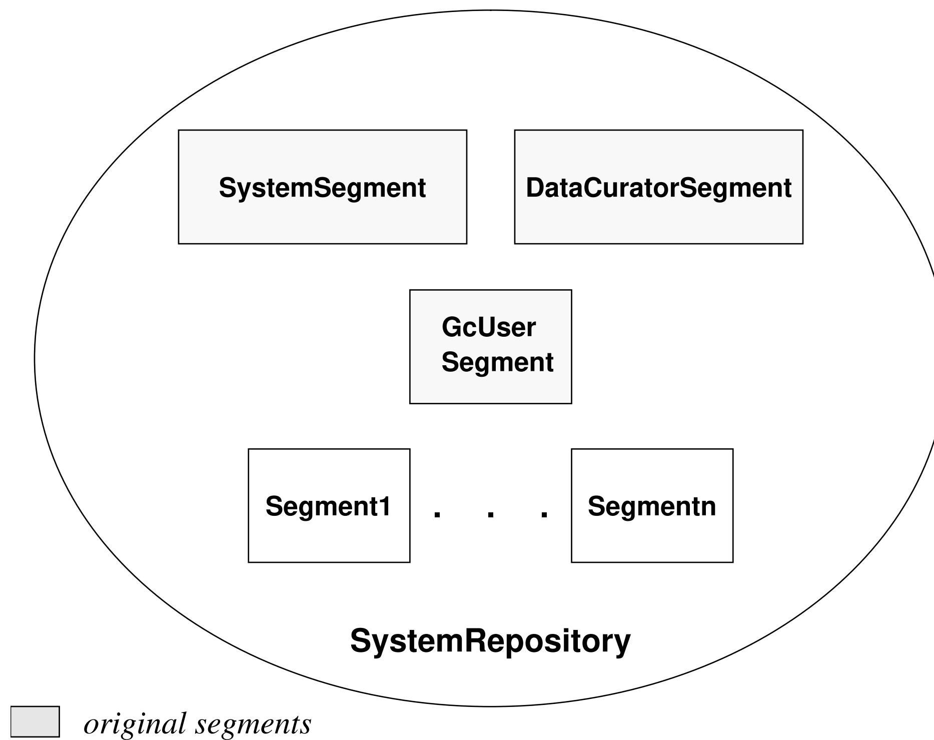
3. **GcUser**

   This segment is used by the system for reclaiming storage.

These segments are shown as part of the Repository in Figure 7.4.

Each segment in the Repository contains the following instance variables: `itsRepository; itsOwner; groupIds;` and `authorizations` (a SmallInteger that indicates whether each group is authorized to read and/or write objects in this segment).

Figure 7.4 Segments in a GemStone repository



Changing the Segment for an Object

If you have the authorization, you can change the accessibility of an individual object by assigning it to a different segment. Class Object defines a message that returns the segment to which the receiver is assigned, and another message that assigns the receiver to a new segment.

The message `segment` returns the segment to which the receiver is assigned:

Example 7.3

```
UserGlobals segment
```

The message `changeToSegment:` *aSegment* assigns the receiver to the segment *aSegment*. You must have write authorization for both segments: the argument and the receiver. Assuming the necessary authorization, this example assigns class Employee to a new segment:

```
Employee changeToSegment: aSegment.
```

You may override the method `changeToSegment:` for your own classes, especially if they have several components.

For objects having several components, such as collections, you may assign all the component objects to a specified segment when you reassign the composite object. You can implement the message `changeToSegment:` *aSegment* to perform these multiple operations. Within the method `changeToSegment:` for your composite class, send the message `assignToSegment:` to the receiver and each object of which it is composed.

For example, a `changeToSegment:` method for the class Menagerie might appear as shown in Example 7.4. The object itself is assigned to another segment using the method `assignToSegment:`. Its component objects, the animals themselves, have internal structure (names, habitats, and so on), and therefore call Animal's `changeToSegment:` method, which in its turn sends the message `assignToSegment:` to each component of anAnimal, ensuring that each animal is properly and completely reassigned to the new segment.

Example 7.4

```
(Array subclass: 'Menagerie'
      instVarNames: #( )
      inDictionary: UserGlobals) name

method: Menagerie
changeToSegment: aSegment
  self assignToSegment: aSegment.
1 to self size do:
      [:eachAnimal | eachAnimal changeToSegment: aSegment. ]
%
```

SmallInteger, Character, Boolean, and *nil* are assigned the SystemSegment and cannot be assigned another segment.

Segment Ownership

Each segment is owned by one user—by default, the user who created it. A segment's owner has control over who can access the segment's objects. As a segment's owner, you can alter your own access rights at any time, even forbidding yourself to read or write objects assigned to the segment.

You might not be the owner of your default segment. To find out who owns a segment, send it the message `owner`. The receiver returns the owner's UserProfile, which you may read, if you have the authorization:

Example 7.5

```
"Return the userId of the owner of the default segment for
the current Session."
| aUserProf myDefaultSeg |
"get default Segment"
myDefaultSeg := System myUserProfile defaultSegment.
"return its owner's UserProfile"
aUserProf := myDefaultSeg owner.
"request the userId"
aUserProf userId

user1
```

Every segment understands the message `owner:` *aUserProfile*. This message assigns ownership of the receiver to the person associated with *aUserProfile*. The following expression, for example, assigns the ownership of your default segment to the user associated with *aUserProfile*:

```
System myUserProfile defaultsegment owner: aUserProfile
```

In order to reassign ownership of a segment, you must have write authorization for the DataCuratorSegment. Because of the way separate authorizations for owners, groups and world combine, changing access rights for the any one of them may not alter a particular user's rights to a segment.

WARNING:
Do not, under any circumstances, attempt to change the authorization of the SystemSegment.

Revoking Your Own Authorization—a Side Effect

You may occasionally want to create objects and then take away authorization for modifying them.

CAUTION:
Do not remove your write authorization for your default segment or your current segment. If lose write authorization for your default segment, you will not be able to log in again. If you lose write authorization for your current segment, Smalltalk execution can halt with an error report after you commit your transaction.

Losing write authorization to your current segment can create a problem because the object server creates and modifies objects assigned to the current segment to keep track of its execution state. If you no longer have write authorization for that segment after committing your transaction, execution must halt because the object server can no longer write those temporary objects.

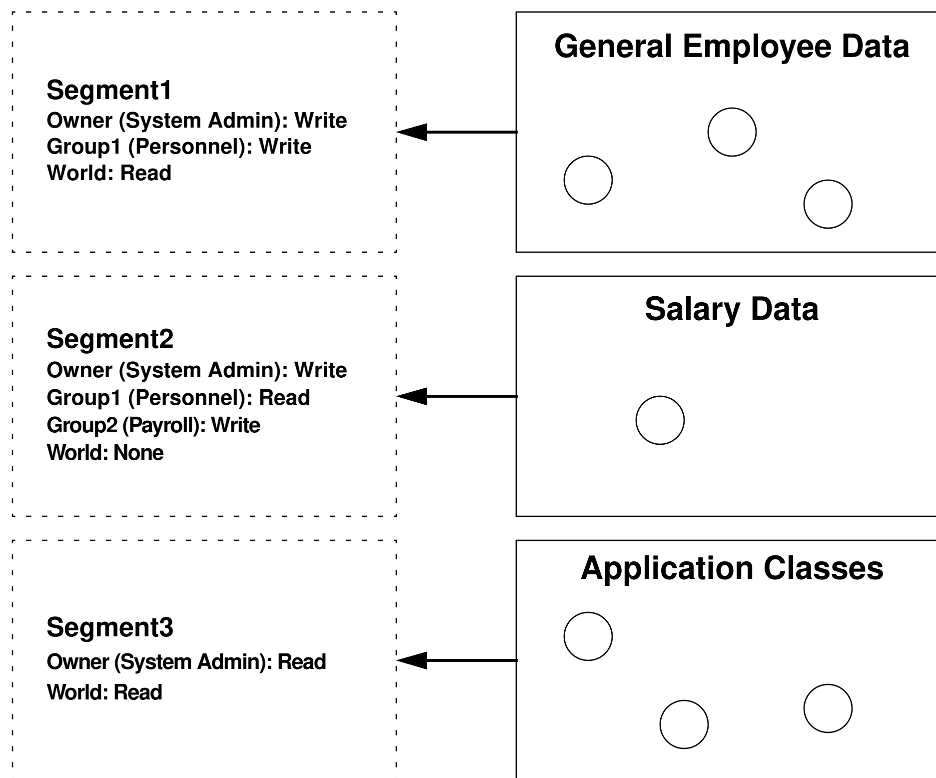
7.2 An Application Example

The structure of the user community determines how your data is stored and accessed. Regardless of their job titles, users generally fall into three categories:

- *Developers* define classes and methods.
- *Updaters* create and modify instances.
- *Reporters* read and output information.

When you have a group of users working with the same GemStone application, you need to ensure that everyone has access to the objects that should be shared, such as the application classes, but you probably want to limit access to certain data objects. Figure 7.5 shows a typical production situation.

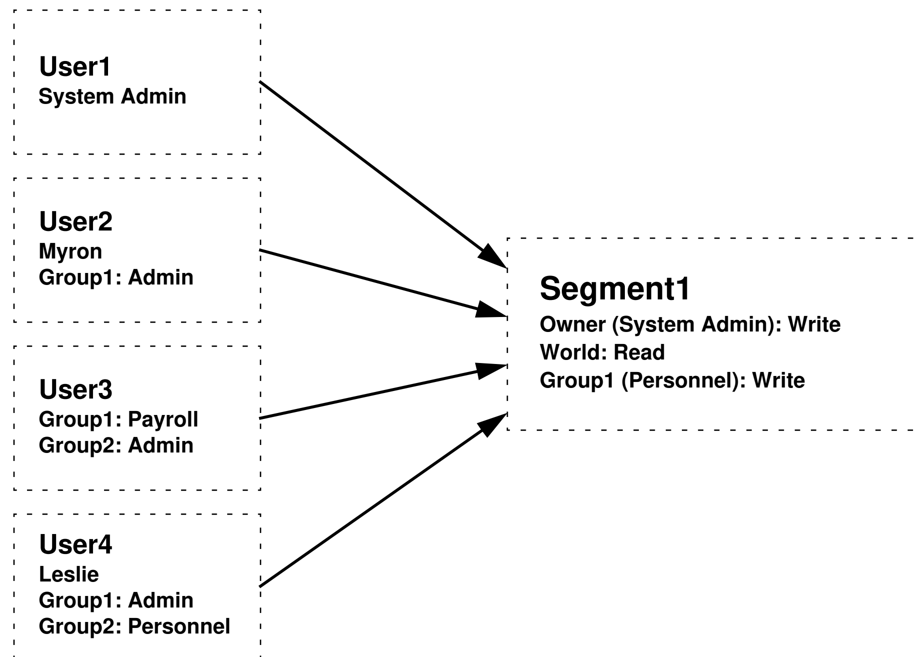
Figure 7.5 Application Objects Assigned to Three Segments



In this example, all the application users need access to the data, but different users need to read some objects and write others. So most data goes into Segment1, which anyone can look at, but only the Personnel group or owner can change. Segment 2 is set up for sensitive salary data, which only the Payroll group or owner can change, and only they and the Personnel group can see. You don't want anyone to accidentally corrupt the application classes, so they go into Segment3, which no one can change.

Look at how the user name, group membership, and segment authorization control access to objects, as shown by Figure 7.6 and Figure 7.7:

Figure 7.6 User Access to Application Segment1

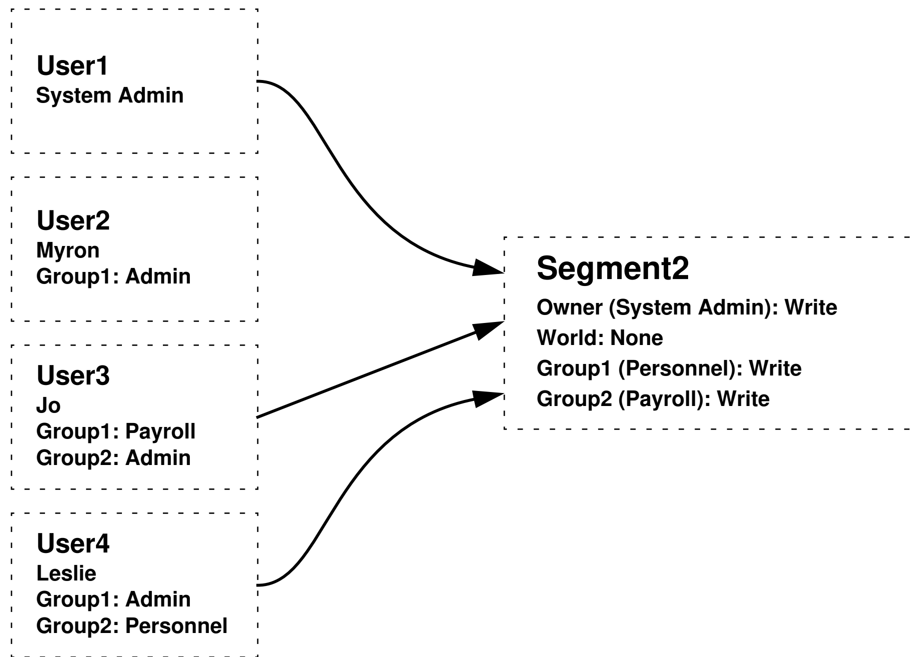


Four users access this application:

- The **System Administrator** owns both segments and can read and write the objects assigned to them.
- **Leslie** belongs to the Personnel group, which authorizes her to read and write Segment1's objects and read Segment2's objects.
- **Jo** can read and write the objects assigned to Segment2, because she belongs to the Payroll group. She doesn't belong to a group that can access Segment1, but she can still read those objects, because Segment1 gives read authorization to all GemStone users.
- **Myron** does not belong to a group that can access either segment. He can read the objects assigned to Segment1 objects, because it allows read access to all GemStone users. He has no access at all to Segment2.

Leslie and Jo are sometimes updaters and sometimes reporters, depending on the type of data. Myron is strictly a reporter.

Figure 7.7 User Access to Application Segment2

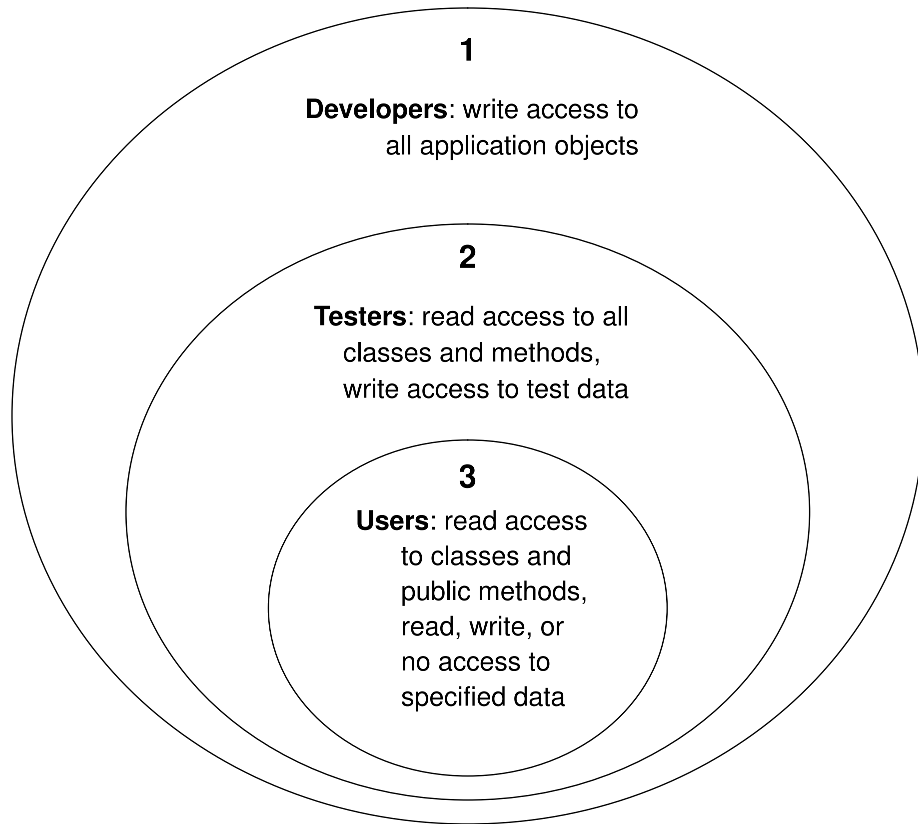


7.3 A Development Example

Up to now, this discussion has been limited to applications in a production environment, but issues of access and security arise at each step of application development. During the design phase you need to consider the segments needed for the application life cycle: development, testing, and production.

The access required at each stage is a subset of the preceding one, as shown in Figure 7.8.

Figure 7.8 Access Requirements During an Application's Life Cycle



Planning Segments for User Access

As you design your application, decide what kind of access different end users will need for each object.

Protecting the Application Classes

All the application users need read access to the application classes and methods, so they can execute the methods. You probably want to limit write access, however, to prevent accidental damage to them. You may even want to change the owner's authorization to read, until changes are required.

The application classes do not require a separate segment. You can assign them to any segment that has read authorization for world and write authorization (or read, if you prefer) for owner. Like other objects, classes and their methods are assigned to segments on an object-by-object basis. Each method can have a different segment, if you want.

Planning Authorization for Data Objects

Authorization for data objects means protecting the instances of the application's classes, which will be created by end users to store their data. You can begin the planning process by creating a matrix of users and their required access to objects. Table 7.1 shows part of such a matrix, which maps out access to instances of the class Employee and some of its instance variables.

Security is easier to implement if it is built into the application design at the beginning, not added later. In the following sections, planning for the third stage, end user access, comes first. Following the planning discussion comes the implementation instructions, which explain how to set up segments for the developers, extend the access to the testers, and finally move the application into production.

Remember that in effect you have four options, shown on the matrix as:

W — need to write (also allows reading)

R — need to read, must not write

N — must not read or write

blank — don't need access, but it won't hurt

Table 7.1 Access for Application Objects Required by Users

	Users						
Objects	**System Admin.**	**Human Resource**	**Employee Records**	**Payroll**	**Mktg**	**Sales**	**Customer Support**
anEmployee	W	W	W	R	R	R	R
name	W	W	W	R	R	R	R
position	W	W	W	R		R	
dept.	W	W	W	R		R	
manager	W	W	W	R		R	
dateHired	W	W	W	R	N	R	N
salary	W	R	R	W	N	N	N
salesQuarter	W	R	R	R	N	W	N
salesYear	W	R	R	R	N	W	N
vacationDays	W	W	W	N	N	N	N
sickDays	W	W	W	N	N	N	N

World Access

To begin analyzing your access requirements, check whether the objects have any Ns. For objects that do, world authorization must be set to none.

If you have people who need read access to nonsensitive information, give world read authorization to those objects. In this example, world can have read access to anEmployee, name, position, dept., and manager. The objects can still be protected from casual browsing by storing them in a dictionary that does not appear in everyone's symbol list. This does not absolutely prevent someone from finding an object, but it makes it difficult. For more information, see Chapter 3, "Name Resolution and Object Sharing".

Owner

By default, the owner has write access to the objects in a segment. To choose an owner, look for a user who needs to modify everything. In terms of the basic user categories described earlier, the owner could be either an administrator or an update. This depends on the type of objects that will be assigned to the segment.

In Table 7.1 the system administrator is the user who needs write access. So the system administrator is made the owner, with full control of all the objects. The

DataCurator and SystemUser logins are available to the system administrator. The DataCurator is not automatically authorized to read and write all objects, however. Like any other user account, it must be explicitly authorized to access objects in segments it does not own. Although the SystemUser can read and write all objects, it should not be used for these purposes.

Planning Groups

The rest of the access requirements must be satisfied by setting up groups. The thing to remember about groups is that they do not reflect the organization chart; they reflect differences in access requirements. Because the number of possible authorization combinations is limited, the number of groups required is also limited.

First look at the existing access to anEmployee, name, position, dept., and manager, as shown in Table 7.2. By making the system administrator the owner with write authorization and assigning read authorization to world, you have already satisfied the needs of five departments.

Table 7.2 Access to the First Five Objects Through Owner and World Authorization

	Users						
Objects	System Admin.	Human Resource	Employ. Records	Payroll	Mktg.	Sales	Customer Support
Employee	W	W	W	R		R	
name	W	W	W	R		R	
position	W	W	W	R		R	
dept.	W	W	W	R		R	
manager	W	W	W	R		R	

☐ write access as owner or read access as world

You still need to provide authorization for the Human Resources and Employee Records departments. In every case, they need the same access (see Table 7.1) so you only have to create one group for the two departments. This group, named Personnel, requires write authorization for the objects in Table 7.2.

Now look at the existing access to the rest of the objects. These objects store more sensitive information, so access requirements of different users are more varied. Assigning write authorization to owner and none to world has completely satisfied the needs of three departments, as shown in Table 7.3.

Table 7.3 Access to the Last Six Objects Through Owner and World Authorization

	Users						
Objects	**System Admin.**	**Human Resource**	**Employ. Records**	**Payroll**	**Mktg**	**Sales**	**Customer Support**
dateHired	W	W	W	R	N	R	N
salary	W	R	R	W	N	N	N
salesQuarter	W	R	R	R	N	W	N
salesYear	W	R	R	R	N	W	N
vacationDays	W	W	W	N	N	N	N
sickDays	W	W	W	N	N	N	N

write access as owner or no access as world

Two more departments, Human Resources and Employee Records, are already set up to access as the Personnel group. As shown in Table 7.4, this group needs write authorization to dateHired, vacationDays, and sickDays, which they must be able to read and modify. They need read authorization to salary, salesQuarter, and salesYear, which they must read but cannot modify.

Table 7.4 Access to the Last Six Objects Through the Personnel Group

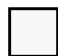
	Users						
Objects	**System Admin.**	**Human Resource**	**Employ. Records**	**Payroll**	**Mktg**	**Sales**	**Customer Support**
dateHired	W	W	W	R	N	R	N
salary	W	R	R	W	N	N	N
salesQuarter	W	R	R	R	N	W	N
salesYear	W	R	R	R	N	W	N
vacationDays	W	W	W	N	N	N	N
sickDays	W	W	W	N	N	N	N

read or write access as Personnel group

Now the Payroll and Sales departments still require access to the objects, as shown in Table 7.3. Because these departments' needs don't match anyone else's, they must each have a separate group.

Table 7.5 Access to the Last Six Objects Through the Payroll and Sales Groups

	Users						
Objects	**System Admin.**	**Human Resource**	**Employ. Records**	**Payroll**	**Mktg**	**Sales**	**Customer Support**
dateHired	W	W	W	R	N	R	N
salary	W	R	R	W	N	N	N
salesQuarter	W	R	R	R	N	W	N
salesYear	W	R	R	R	N	W	N
vacationDays	W	W	W	N	N	N	N
sickDays	W	W	W	N	N	N	N

 read or write access as Payroll or Sales group

In all, this example only requires three groups: Personnel, Payroll, and Sales, even though it involves seven departments.

Planning Segments

When you have been through this exercise with all your application's prospective objects and users, you are ready to plan the segments. For easiest maintenance, use the smallest number of segments that your required combinations of owner, group, and world authorizations allow. You don't need different segments with duplicate functionality to separate particular objects, like the application classes and data objects. Remember that symbol lists, not segments, are used to organize objects for listing and retrieval.

In this example you need six segments, as shown in Figure 7.9. Notice that each one has different authorization.

Developing the Application

During application development you implement two separate schemes for object organization: one for sharing application objects by the development team and one controlling access by the end users. In addition, you may need to allow access for the testers, who may need different access to objects.

Once you have planned the segments and authorizations you want for your project, you can refer to procedures in the *GemStone System Administration Guide* for implementing that plan.

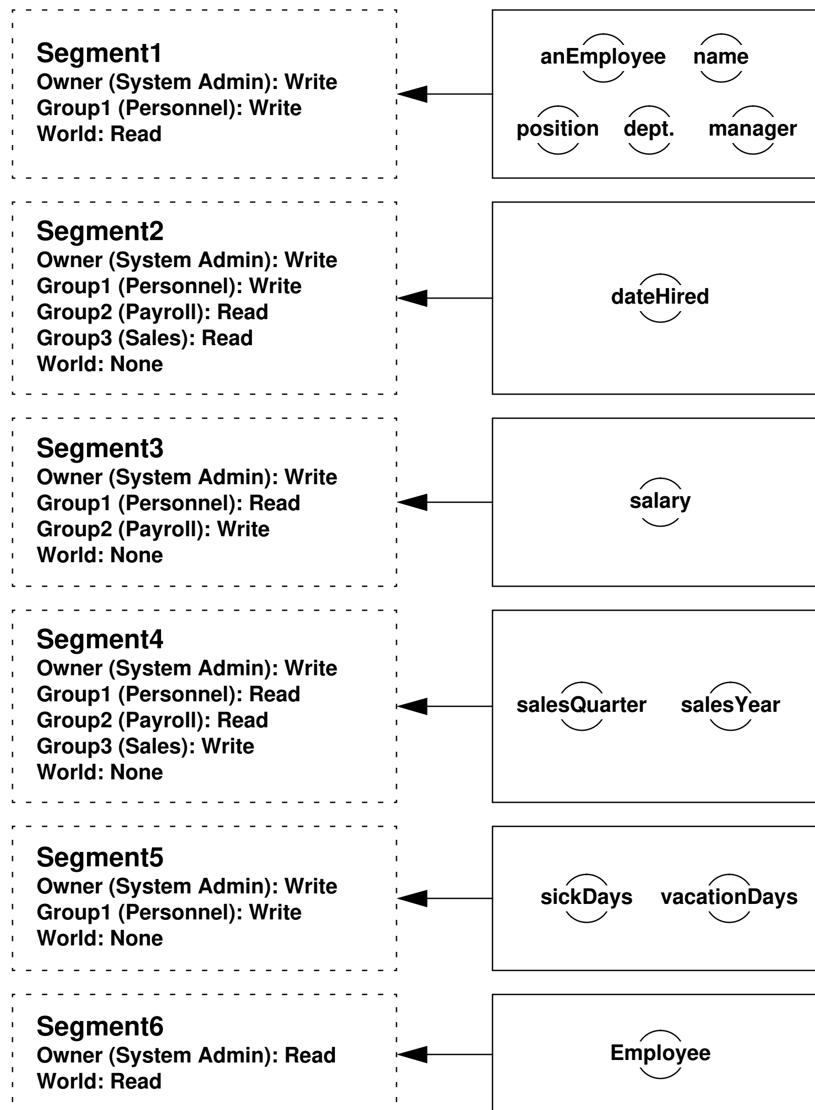
Setting Up Segments for Joint Development

To make joint development possible, you need to set up authorization and references so that all the developers have access to the classes and methods that are being created. Create a new symbol dictionary for the application and put it in everyone's symbol list; make sure it includes references to any shared segments. If only developers are using the repository, you can give world access to shared objects, but if other people are using the repository, you must set up a group for developers.

You can organize segment assignments in various ways:

- **Full access to all personal segments.** Give all the developers their own default segments to work in. Give everyone in the team write access to all the segments. Because the objects you create are typically assigned to your default segment, this method may be the simplest way to organize shared work.
- **Read access to all personal segments.** Set up the same as above, except give everyone read access to the segments. If each developer is doing a separate module, read access may be enough. Then everyone can use other people's classes, but not change them. This has the advantage of enforcing the line between application and data.
- **Full access to a shared segment.** Give all developers the same default segment, writable by everyone. This is an easy, informal way to share objects.
- **Full access to a shared segment plus private segments.** Developers work in their own default segments and reassign their objects to the shared segment when they are finished. This lets you share a collection, for example, but keep the existing elements private, so that other developers could add elements but not modify the elements you have already created. To share a collection this way, assign the collection object itself to the accessible segment. The collection has references to many other objects, which can be associated with other segments. Everyone has the references, but they get errors if they try to access objects assigned to non-readable segments. You might also choose to share an application symbol dictionary, so that other developers can put objects in it, without making the objects themselves public.

Figure 7.9 Segments Required for User Access to Application Objects



Making the Application Accessible for Testing

Testers need to be able to alternate between two distinct levels of access:

- **Full access.** As members of the development team, they need read access to all the classes and methods in the application, including the private methods. Testers also need write access to their test data.
- **User-level access.** They need a way to duplicate the user environment, or more likely several environments created for different user groups.

This can be done by setting up a tester group and one or more sample user groups during the development phase. For testing the user environment, the application must already be set up for multi-user production use, as explained in the following section.

Moving the Application into a Production Environment

When you have created the application, it is time to set it up for a multi-user environment. A GemStone application is developed in the repository, so all you have to do to install an application is to give other users access to it. This means implementing the rest of your application design, in roughly the reverse order of the planning exercise. To give other users authorization to use the objects in the application:

1. Create the segments.
2. Create the necessary user groups specified in up-front development, if they don't exist.
3. Assign the required owner, world, and group authorizations to the segments.
4. Assign testers to the user groups and complete multi-user testing.
5. Assign any end users that need group authorization to the user groups.
6. Assign the application's objects to the segments you created.

You also have to give users a reference to the application so they can find it. An application dictionary is usually created with references to the application objects, including its segments. A reference to this dictionary usually must appear in the users' symbol lists. For more information on the use of symbol dictionaries, see the discussion of symbol resolution and object sharing in Chapter 3, "Name Resolution and Object Sharing."

Segment Assignment for User-created Objects

Because segment assignment is on an object-by-object basis, it is important to know how objects are assigned. When the objects are being created by end users of an application, as in this example, you may want to partially or fully automate the process of segment assignment. Depending on the needs of the local site, you can implement various mechanisms to ensure data security, prevent accidental damage to existing data, or simply avoid misplaced data.

Assign a Specified Segment to the User Account

Set up users with the proper application segment by default. This is a simple way to assure that someone who creates objects in a single application segment doesn't misplace them. To make it impossible to change segments, rather than just unlikely, you also have to close write access for group and world to all the other segments.

This solution would work for the Sales and Payroll groups in the example (Figure 7.9 on page 7-27). They need read access to several segments, but they only write in one.

The drawback of this solution is that the user can only use one application.

Develop the Application to Create the Data Objects

Your best choice is to create objects in the correct segment, using the `Segment setCurrentWhile:` method. With this method, the application stores data objects in the proper segments. This provides the most protection. Besides guaranteeing that the objects end up in the proper segment, this prevents users from accidentally modifying objects they have created. It also prevents them from reading the data that other users enter, even when everyone is creating instances of the same classes.

7.4 Assigning Objects to Segments

For segment authorizations to have any effect, you must assign some objects to the segments whose authorizations you have set up.

Segments for New Objects

Your UserProfile stores a reference to a segment that serves as your *default segment*. This is the segment to which GemStone normally assigns the new objects you create.

Removing Segments

Every object assigned to a segment refers to that segment. Because potentially many objects can refer to any segment, removing a segment is a more complex matter than simply executing one method to remove it from the SystemRepository. You must first decide whether you wish to remove all the objects assigned to it, or whether you wish to reassign some or all of them to another segment. As long as an object that you or another user can refer to within your application is assigned to the segment, GemStone cannot reclaim the storage used by the segment.

Removing a segment, therefore, is a somewhat delicate task related to the manner in which GemStone reclaims storage. It is a four-step process:

1. Determine which segment you wish to remove.
2. Determine which objects refer to that segment.
3. Remove those objects or assign them to another segment, as required.
4. Remove the segment.

The details of accomplishing this are as follows:

1. Segments are stored in the indexed instance variables of SystemRepository. Inspect the SystemRepository to determine the index of the segment you wish to remove.
2. Segments do not know which objects refer to them, but objects know which segment they refer to. You can determine an object's segment assignment by sending the object the message `segment`. Printing the result provides a reference to the segment. Determine the objects assigned to the target segment using this or other mechanisms you may have built into your application.
3. Using `changeToSegment:` (implemented in class Object and reimplemented in a few subclasses), reassign objects into other segments if you need to. If you wish to remove the objects instead, remove all references to the objects.
4. When no objects that you wish to keep refer to the segment, it is safe to remove it. To do so, execute an expression of the form:

   ```
   SystemRepository deleteObjectAt: 5
   ```

   The argument to the message `deleteObjectAt:` must be the index of the segment you wish to delete, as determined by the inspection of SystemRepository you performed in the first step, above.

5. If you still have a reference to that segment through an object, it is also possible to execute an expression of the form:

   ```
   SystemRepository removeValue: (anObject segment)
   ```

   The argument to the message `removeValue:` must be a pointer to the segment you wish to delete.

After you commit your changes, the segment is available for storage reclamation.

7.5 Privileged Protocol for Class Segment

Privileges stand apart from the segment and authorization mechanism. *Privileges* are associated with certain operations: they are a means of stating that, ordinarily, only the DataCurator or SystemUser is to perform these privileged operations. The DataCurator can assign privileges to other users at his or her discretion, and then those users can also perform the operations specified by the particular privilege.

NOTE
Privileges are more powerful than segment authorization. Although the owner of a segment can always use read/write authorization protocol to restrict access to objects in a segment, the DataCurator can override that protection by sending privileged messages to change the authorization scheme.

The following message to Segment always requires special privileges:

```
newInRepository:             (class method)
```

You can always send the following messages to the segments you own, but you must have special privileges to send them to other segments:

```
group:authorization:
ownerAuthorization:
worldAuthorization:
```

For changing privileges, UserProfile defines two messages that also work in terms of the privilege categories described above. The message `addPrivilege:` *aPrivString* takes a number of strings as its argument, including the following:

```
'DefaultSegment'
'SegmentCreation'
'SegmentProtection'
```

To add segment creation privileges to your UserProfile, for example, you might do this:

```
System myUserProfile addPrivilege: 'SegmentCreation'.
```

This gives you the ability to execute `Segment newInRepository: SystemRepository`.

A similar message, `privileges:`, takes an array of privilege description strings as its argument. The following example adds privileges for segment creation and password changes:

```
System myUserProfile privileges:
                    #('SegmentCreation' 'UserPassword')
```

To withdraw a privilege, send the message `deletePrivilege:` *aPrivString*. As in preceding examples, the argument is a string naming one of the privilege categories. For example:

```
System myUserProfile deletePrivilege: 'SegmentCreation'
```

Because UserProfile privilege information is typically stored in a segment that only the data curator can modify, you might not be able to change privileges yourself. You must have write authorization to the DataCuratorSegment in order to do so.

For direction and information about configuring user accounts, adding user accounts and assigning segments to those accounts, and checking authorization for user accounts, see the *GemStone System Administration Guide*.

7.6 Segment-related Methods

Most of the methods used for basic operations on segments are implemented in the GemStone kernel class Segment. For the protocol of class Segment, see the *GemStone Kernel Reference*. Methods for segment-related operations are also implemented in a few other classes:

Class

Instance Protocol: Authorization

`changeToSegment:` *segment*

Assign the receiver and its non-shared components to the given segment. The segments of class variable values are not changed. The current user must have write access to both the old and new segments for this method to succeed.

Object

Instance Protocol: Updating

`assignToSegment:` *aSegment*

Reassigns the receiver to *aSegment*. The user must be authorized to write to both segments (the receiver's current segment and *aSegment*). Generates an error if the repository containing *aSegment* is full, if there is an authorization conflict, or if the receiver is a self-defining object (SmallInteger, AbstractCharacter, Boolean, or UndefinedObject).

`changeToSegment:` *segment*

Assign the receiver to the given *segment*. This method just calls `assignToSegment:` *aSegment*. You can reimplement it, however, to assign components of the receiver as well. This has been done for class Class (above). Use that version as an example for implementations tailored to your own classes.

System

Class Protocol: Session Control

`currentSegment`

Return the Segment in which objects created in the current session are stored. At login, the current segment is the default segment of the UserProfile for the session of the sender.

`currentSegment:` *aSegment*

Redefines the Segment in which subsequent objects created in the current session will be stored. Return the receiver. Exercise caution when executing this method. If, at the time you attempt to commit your transaction, you no longer have write authorization for *aSegment*, an error will be generated, and you will be placed back into your default Segment.

UserProfile

Instance Protocol: Accessing

`defaultSegment`

Return the default login Segment associated with the receiver.

Instance Protocol: Updating

`defaultSegment:` *aSegment*

Redefines the default login Segment associated with the receiver, and return the receiver.

This method requires the #DefaultSegment privilege.

You must have write authorization for the Segment where the UserProfile resides. Exercise extreme caution when using this method. If, at the time you commit your transaction, the receiver no longer had write authorization for *aSegment*, that user's GemStone session will be terminated and the user will be unable to log back in to GemStone.

Class Protocol: Instance Creation

`newWithUserId:` *aSymbol* `password:` *aString* `defaultSegment:` *aSegment* `privileges:` *anArrayOfStrings* `inGroups:` *aCollectionOfGroupSymbols*

Return a new UserProfile with the associated characteristics. The default compiler language is ASCII.

`newWithUserId:` *aSymbol* `password:` *aString* `defaultSegment:` *aSegment* `privileges:` *anArrayOfStrings* `inGroups:` *aCollectionOfGroupSymbols* `compilerLanguage:` *aLangString*

Creates a new UserProfile with the associated characteristics. In so doing, creates a symbol list with two dictionaries: UserGlobals, Globals, and Published. The first Dictionary (UserGlobals) is created for the user's private symbols, and initially contains a single Association whose key is UserGlobals and whose value is the dictionary itself. Return the new UserProfile.

Before the new user may log in to GemStone, the new UserProfile must be added to the UserProfileSet AllUsers, and the user must be authorized to read and write in the specified default Segment.

UserProfileSet

Instance Protocol: Adding

If the receiver is not AllUsers, a new user will be unable to log in to GemStone. In addition, in order to log into GemStone, a user must be authorized to read and write in the default Segment that is specified for that user.

`addNewUserWithId:` *aSymbol* `password:` *aPassword*

Creates a new UserProfile and adds it to the receiver. The new UserProfile has no privileges, and belongs to no groups. This method creates a new Segment, which is owned by the new user and assigned as the user's default segment. The new Segment is created with world-read permission.

This default method can be used by the data curator in batch user installations. Return the new UserProfile.

If the receiver is not AllUsers, the new user will be unable to log in to GemStone.

`addNewUserWithId:` *aSymbol* `password:` *aString* `defaultSegment:` *aSegment* `privileges:` *anArrayOfStrings* `inGroups:` *aCollectionOfGroupSymbols*

Creates and return a new UserProfile with the associated characteristics, and adds it to the receiver. Generates an error if the userId *aSymbol* duplicates the userId of any existing element of the receiver.

`addNewUserWithId:` *aSymbol* `password:` *aString* `defaultSegment:` *aSegment* `privileges:` *anArrayOfStrings* `inGroups:` *aCollectionOfGroupSymbols* `compilerLanguage:` *aLangString*

Creates a new UserProfile with the associated characteristics and adds it to the receiver. Generates an error if the userId *aSymbol* duplicates the userId of any existing element of the receiver. Return the new UserProfile.

If the receiver is not AllUsers, the new user will be unable to log in to GemStone. In addition, in order to log in to GemStone, the user must be authorized to read and write in the specified default Segment.

Chapter 8

Class Versions and Instance Migration

Few of us can design something perfectly the first time. Although you undoubtedly designed your schema with care and thought, after using it for a while you will probably find a few things you would like to improve. Furthermore, the world seldom remains the same for very long. Even if your design was perfect, real-world changes usually require changes to the schema sooner or later. This chapter discusses the mechanisms GemStone Smalltalkprovides to allow you to make these changes.

Versions of Classes
: defines the concept of a class version and describes two different approaches you can take to specify one class as a version of another.

ClassHistory
: describes the Smalltalk class that encapsulates the notion of class versioning.

Migrating Objects
: explains how to migrate either certain instances, or all of them, from one version of a class to another while retaining the data that these instances hold.

8.1 Versions of Classes

You cannot create instances of modifiable classes. In order to create instances—in other words, in order to populate your database with usable data—you defined your classes as well as you could, and then, when you believed that your schema was fully defined, you sent the message `immediateInvariant` to your classes. They were thereafter no longer modifiable, and instances of them could be created. You may now have instances of invariant classes populating your database and a need to modify your schema by redefining certain of these classes.

To support this inevitable need for schema modification, GemStone allows you to define different versions of classes. Every class in GemStone has a class history—an object that maintains a list of all versions of the class—and every class is listed in exactly one class history. You can define as many different versions of a class as required, and declare that the different versions belong to the same class history. You can migrate some or all instances of one version of a class to another version when you need to. The values of the instance variables of the migrating instances are retained, if you have defined the new version to do so.

NOTE
Although this chapter discusses schema migration in the context of GemStone Smalltalk, the various interfaces have tools to make the job easier. The functionality described in this chapter is common to all interfaces. Consult your GemBuilder manual for other ways in which you might lighten your burden.

Defining a New Version

In GemStone Smalltalk classes have *versions*. Each version is a unique class object, but the versions are related to each other through a common class history. The classes need not share a similar structure, nor even a similar implementation. The classes need not even share a name, although it is probably less confusing if they do, or if you establish and adhere to some naming convention.

You can take one of two approaches to defining a new version:

- Define a class having the same name as an existing class. The new class automatically becomes a new version of the previous class. Existing instances remain unchanged, and have access to the old class's methods. Instances created after the redefinition have the new class's structure and access to the new class's methods.
- Define a new class by another name, and then declare explicitly that it shares the same class history as the original class. You can do this with any of the class creation messages that include the keyword `newVersionOf:`.

8.2 ClassHistory

In GemStone Smalltalk, any class can be associated with a class history, represented by the system as an instance of the class ClassHistory. A class history is an array of classes that are meant to be different versions of each other.

Defining a Class with a Class History

When you define a new class whose name is the same as an existing class in one of your symbol list dictionaries, it is by default created as the latest version of the existing class and shares its class history.

When you define a new class by a name new to your symbol list dictionaries, it is by default created with a unique class history. If you use a class creation message that includes the keyword `newVersionOf:`, you can specify an existing class whose history you wish the new class to share.

For example, the following expression creates a class named NewAnimal and specifies that the class shares the class history that the existing class Animal uses. This action has the effect of renaming the class:

Example 8.1

```
Object subclass: 'NewAnimal'
   instVarNames: #('diet' 'favoriteFood' 'habitat' 'name'
                   'predator')
   classVars: #()
   classInstVars:  #()
   poolDictionaries:  #[]
   inDictionary: UserGlobals
   constraints: #[]
   instancesInvariant: false
   newVersionOf: Animal
   description: nil
   isModifiable: false
```

The example below installs the newly defined class Animal as a later version of the previously defined class Animal:

Example 8.2

```
Object subclass: 'Animal'
      instVarNames: #('habitat' 'name' 'predator')
      classVars: #()
      classInstVars:  #()
      poolDictionaries:  #[]
      inDictionary: UserGlobals
      constraints: #[]
      instancesInvariant: false
      description: nil
      isModifiable: false
```

If you wish to define a new class Animal with its own unique class history, you can add it to a different symbol list dictionary, and specify the argument *nil* to the keyword `newVersionOf:`.

Example 8.3

```
Object subclass: 'Animal'
      instVarNames: #('habitat' 'name' 'predator')
      classVars: #()
      classInstVars:  #()
      poolDictionaries:  #[]
      inDictionary: MoreUserGlobals
      constraints: #[]
      instancesInvariant: false
      newVersionOf: nil
      description: nil
      isModifiable: false
```

If you try to define a new class with the same name as an existing class you did not create, you will most likely get an authorization error, because you will be trying to modify the class history of that class, an object which you are probably not authorized to modify. If this restriction becomes a problem, use a subclass creation message that includes the keyword `newVersionOf:`, and set it to nil. In this way, the existing class history remains unmodified and your new class has its own class history.

CAUTION:
If you try to define a new class with the same name as one of the GemStone Smalltalk kernel classes, you will definitely get such a write authorization error. Do not use the above workaround in this case. Redefining a kernel class can cause aberrant system behavior and even system failure.

Accessing a Class History

You can access the class history of a given class by sending the message `classHistory` to the class. For example, the following expression returns the class history of the class Employee:

```
Employee classHistory
```

You can use an expression such as this to collect all instances of any version of a class, as you will see in a later example.

Assigning a Class History

You can assign a class history by sending the message `addNewVersion:` to the class whose class history you wish to use, with the class whose history is to be reassigned as the argument. For example, suppose that, when we created the class NewAnimal, we intended to assign it the same class history as Animal, but forgot to do so. To specify that it is a new version of Animal, we execute the following expression:

```
Animal addNewVersion: NewAnimal
```

Class Histories and Constraints

Class histories are also used to determine which classes can satisfy instance variable constraints. When you define an instance variable, you can constrain it to be an instance of a specific class. The class Employee, for example, defined three instance variables—*name*, *job*, and *address*—all of which were constrained to be instances of class String. A fourth instance variable, *age*, was constrained to be an instance of Integer.

Suppose that an address needs more structure than a simple string. To provide this structure, you define a new class called Address. Instances of Address contain variables named *street*, *city*, *state*, and *postalCode*. You will redefine Employee so that its instance variable *address* is constrained to be an instance of Address.

Having defined the class Address, you now want to use it for customer mailings as well as employees' addresses. However, the new class Address has notable omissions. It works well enough for employees, perhaps, but for customers it may need international address information, or it may need to allow for addresses with post office boxes instead of street addresses. You must define a new version of the class Address to include these subtleties, and ensure that it shares the same class history as the previous version.

Having done so, you need not redefine Employee. Its constraint on the instance variable *address* will be satisfied by any object whose class shares a class history with Address. In general, instance variable constraints are satisfied by the class specified by the constraint, any subclass of that class, any class that shares the same class history as one of those classes, or any of its subclasses.

It is possible to define two classes having the same name that do not share the same class history. If you define a new class named Address, for example, that does not share a class history with the old class, and you then attempt to store an instance of the new class Address in the variable *address* of an instance of Employee, you will get a GemStone error that reads (in part):

```
Attempt to store an object of class Address in an instance
variable constrained to hold objects of class Address.
```

Because the two classes have the same name, the error can appear mystifying. However, the solution is simple: either assign the new class Address the same class history as the old one, or use an instance of the old class Address for the employee's *address* instance variable.

How can this situation arise in the first place? How you might define two classes having the same name, but not sharing the same class history, depends upon the particular interface to GemStone you are using and the tool you used to define the subclasses. The most plausible way this situation might arise is when you define the two classes in two different symbol list dictionaries. Chapter 3, "Name Resolution and Object Sharing," describes the purpose and use of symbol list dictionaries.

8.3 Migrating Objects

Once you define two or more versions of a class, you may wish to migrate instances of the class from one version to another. Migration in GemStone Smalltalk is a flexible, configurable operation:

- Instances of any class can migrate to any other, as long as they share a class history. The two classes need not be similarly named, or, indeed, have anything else in common.
- Migration can occur whenever you so specify.
- Not all instances of a class need to migrate at the same time—you can migrate only certain instances at a time. Other instances need never migrate, if that is appropriate.
- The manner in which values of the old instance variables are used to initialize values of the new instance variables is also under your control. A default mapping mechanism is provided, which you can override if you need to.

Migration Destinations

If you know the appropriate class to which you wish to migrate instances of an older class, you can set a migration destination for the older class. To do so, send a message of the form:

```
OldClass migrateTo: NewClass
```

This message configures the old class to migrate its instances to become instances of the new class, but only when it is instructed to do so. Migration does not occur as a result of sending the above message.

It is not necessary to set a migration destination ahead of time. You can specify the destination class when you decide to migrate instances. It is also possible to set a migration destination, and then migrate the instances of the old class to a completely different class, by specifying a different migration destination in the message that performs the migration.

You can erase the migration destination for a class by sending it the message `cancelMigration`. For example:

```
OldClass cancelMigration
```

If you are in doubt about the migration destination of a class, you can query it with an expression of the form:

```
MyClass migrationDestination
```

The message `migrationDestination` returns the migration destination of the class, or nil if it has none.

Migrating Instances

A number of mechanisms are available to allow you to migrate one instance, or a specified set of instances, to either the migration destination, or to an alternate explicitly specified destination.

No matter how you choose to migrate your data, however, it is a good idea to migrate data in its own transaction. That is, as part of preparing for migration, commit your work so far. In this way, if migration should fail because of some error, you can abort your transaction and you will lose no other work; your database will be in a consistent state from which you can try again. After migration succeeds, commit your transaction again before you do any further work. Again, this technique ensures a consistent database from which to proceed.

Finding Instances and References

To prepare for instance migration, two methods are available to help you find instances of specified classes or references to such instances. An expression of the form:

```
SystemRepository listInstances: anArray
```

takes as its argument an array of class names, and returns an array of sets. The contents of each set consists of all instances whose class is equal to the

corresponding element in the argument *anArray*. Instances to which you lack read authorization are omitted without notification.

NOTE:
The above method searches the database once for all classes in the array. Executing `allInstances` *for each class requires searching the database once per class.*

An expression of the form:

```
SystemRepository listReferences: anArray
```

takes as its argument an array of objects, and returns an array of sets. The contents of each set consists of all instances that refer to the corresponding element in the argument *anArray*. Instances to which you lack read authorization are omitted without notification.

Using the Migration Destination

The simplest way to migrate an instance of an older class is to send the instance the message `migrate`. If the object is an instance of a class for which a migration destination has been defined, the object becomes an instance of the new class. If no destination has been defined, no change occurs.

The following series of expressions, for example, creates a new instance of Animal, sets Animal's migration destination to be NewAnimal, and then causes the new instance of Animal to become an instance of NewAnimal.

Example 8.4

```
| aLemming |
aLemming := Animal new.
Animal migrateTo: NewAnimal.
aLemming migrate.
```

Other instances of Animal remain unchanged until they, too, receive the message to migrate.

If you have collected the instances you wish to migrate into a collection named allAnimals, execute:

```
allAnimals do: [:each | each migrate]
```

Bypassing the Migration Destination

You can bypass the migration destination, if you wish, or migrate instances of classes for which no migration destination has been specified. To do so, you can specify the destination directly in the message that performs the migration. Two methods are available to do this—`migrateInstances:to:`, and `migrateInstancesTo:`. Neither of these messages change the class's persistent migration destination. Instead, they specify a one-time-only operation that migrates the specified instances, or all instances, to the specified class, ignoring any migration destination set for the class.

The message `migrateInstances:to:` takes a collection of instances as the argument to the first keyword, and a destination class as the argument to the second. For example:

```
Animal migrateInstances: #[aDugong, aLemming] to: NewAnimal.
```

The example above migrates the specified instances of Animal to instances of NewAnimal.

The message `migrateInstancesTo:` migrates all instances of the receiver to the specified destination class. For example:

```
Animal migrateInstancesTo: NewAnimal.
```

The example above migrates all instances of Animal to instances of NewAnimal.

The following example uses `migrateInstances:to:` to migrate all instances of all versions of a class, except the latest version, to the latest version:

Example 8.5

```
| animalHist allAnimals |
animalHist := Animal classHistory.
allAnimals := SystemRepository listInstances: animalHist.
"Returns an array of the same size as the class history.
 Each element in the array is a set corresponding to one
 version of the class.  Each set contains all the
 instances of that version of the class."

1 to: animalHist size-1 do: [:index | (animalHist at: index)
      migrateInstances:(allAnimals at: index)
      to: (animalHist at: animalHist size)].
```

Migration Errors

Several problems can occur with migration:

- you may be trying to migrate an object that the interpreter needs to remain in a constant state (migrating to self);
- you may be trying to migrate an instance that is indexed, or participates in an index;
- you may lack authorization to read or modify the data.

Errors also occur when two versions of a class have incompatible constraints on their elements or instance variables. This error is explored more fully in the next section, "Instance Variable Mappings" on page 8-13.

Migrating self

Sometimes a requested migration operation can cause the interpreter to halt and display an error message saying: "The object you are trying to migrate was already on the stack." This error occurs when you try to send the message `migrate` (or one of its variants) to *self*. Migration can change the structure of an object. If the interpreter was already accessing the object whose structure you are trying to change, the database can become corrupted. To avoid this undesirable consequence, the interpreter checks for the presence of the object in its stack before trying to migrate it, and notifies you if it finds it.

If you receive such a notifier, you have several options:

- If you have explicitly sent the message `migrate` (or one of its variants) to *self*, rewrite the method in which you have done so to accomplish its purpose in some other manner.
- If you were unaware that you sent the message `migrate` (or one of its variants) to *self*, finish your work with the object in its older form and commit the transaction. Then migrate the object in a new transaction.

> *NOTE:*
> *In order to avoid generating an error, do not send the message* `migrate` *to* self. *The compiler and interpreter cannot work properly if the class of self changes during the execution of a method.*

Migrating Instances That Participate in an Index

If an instance participates in an index (for example, because it is part of the path on which that index was created), then the indexing structure can, under certain circumstances, cause migration to fail. Three scenarios are possible:

- Migration succeeds. In this case, the indexing structure you have made remains intact. Commit your transaction.
- GemStone examines the structures of the existing version of the class and the version to which you are trying to migrate, and determines that migration is incompatible with the indexing structure. In this case, GemStone raises an error notifying you of the problem, and migration does not occur.

  You can commit your transaction, if you have done other meaningful work since you last committed, and then follow these steps:

  1. Remove the index in which the instance participates.
  2. Migrate the instance.
  3. Re-create the index, this time using a constraint appropriate to the new class version.
  4. Commit the transaction.
- In the final case, GemStone fails to determine that migration is incompatible with the indexing structure, and so migration occurs and the indexing structure is corrupted. In this case, GemStone raises an error notifying you of

the problem, and you will not be permitted to commit the transaction. Abort the transaction and then follow the steps explained above.

NOTE
To avoid loss of work in case you must abort the transaction, always commit your transaction before you begin data migration.

For more information about indexing, see Chapter 5, “Querying.” For more information about committing and aborting transactions, see Chapter 6, “Transactions and Concurrency Control.”

Migration and Authorization

You cannot migrate instances that you are not authorized to read or write. The migration methods `migrateInstancesTo:` and `migrateInstances: to:` return an array of four collections:

- objects to which you lack read authorization;
- objects to which you lack write authorization;
- objects that are instances of indexed collections; and
- objects whose class was not identical to the receiver—presumably, incorrectly gathered instances.

If all four of these collections are empty, all requested migrations have occurred.

Instance Variable Mappings

Earlier, we explained that migration can involve changing the structure of an object. By now, you are probably wondering what happens to the values of the variables in that object—the class, class instance, and instance variables.

When an object is migrated, it refers to the class and class instance variables that have been defined for the new version of the class. These variables have whatever values have been assigned to them in the class object.

Migrating instances, however, is not terribly helpful unless you can retain the data they contain. Instance variables, therefore, can retain their values when you migrate instances. The following discussion describes the default manner in which instance variables are mapped. This default arrangement can be modified if necessary.

Default Instance Variable Mappings

The simplest way to retain the data held in instance variables is to use instance variables with the same names in both class versions. If two versions of a class have instance variables with the same name, then the values of those variables are automatically retained when the instances migrate from one class to the other.

Suppose, for example, you create two instances of class Animal and initialize their instance variables as shown:

Example 8.6

```
| aLemming aDugong |
aLemming := Animal new.
aLemming name: 'Leopold'.
aLemming favoriteFood: 'grass'.
aLemming habitat: 'tundra'.
aDugong := Animal new.
aDugong name:  'Maybelline'.
aDugong favoriteFood: 'seaweed'.
aDugong habitat: 'ocean'.
```

You then decide that class Animal really needs an additional instance variable, *predator*, which is a Boolean—*true* if the animal is a predator, *false* otherwise. You create a class called NewAnimal, and define it to have four instance variables: *name*, *favoriteFood*, *habitat*, and *predator*, creating accessing methods for all four. You then migrate aLemming and aDugong. What values will they have?

Example 8.7 takes the class and method definitions for granted and performs the migration. It then shows the results of printing the values of the instance variables.

Example 8.7

```
| bagOfAnimals |
bagOfAnimals := IdentityBag new.
bagOfAnimals add: aLemming; add: aDugong.
Animal migrateInstances: bagOfAnimals to: NewAnimal.
aLemming name.
Leopold

aLemming favoriteFood.
grass

aLemming habitat.
tundra

aLemming predator.
nil

aDugong name.
Maybelline

aDugong favoriteFood.
seaweed

aDugong habitat.
ocean

aDugong predator.
nil
```

As you see, the migrated instances retained the data they held. They have done so because the class to which they migrated defined instance variables that had the same names as the class from which they migrated. The new instance variable *name* was initialized with the value of the old instance variable *name*, and so on.

The new class also defined an instance variable, *predator*, for which the old class defined no corresponding variable. This instance variable therefore retains its default value of *nil*.

If the class to which you migrate instances defines no instance variable having the same name as that of the class from which the instance migrates, the data is dropped. For example, if you migrated an instance of NewAnimal back to become an instance of the original Animal class, any value in *predator* would be lost. Because Animal defines no instance variable named predator, there is no slot in which to place this value.

To summarize, then:

- If an instance variable in the new class has the same name as an instance variable in the old class, it retains its value when migrated.
- If the new class has an instance variable for which no corresponding variable exists in the old class, it is initialized to *nil* upon migration.
- If the old class has an instance variable for which no corresponding variable exists in the new class, the value is dropped and the data it represents is no longer accessible from this object.

Customizing Instance Variable Mappings

Three different kinds of customization are available to accommodate three different situations.

- In one situation, you may wish to initialize a variable having one name with the value of a variable having a different name.
- In another situation, you may wish to perform a specific operation on the value of a given variable before initializing the corresponding variable in the class to which the object is migrating.
- You may also have a situation in which the only change to a variable is the class of which it is constrained to be an instance. If the new constraint class is not a subclass of the old constraint class, a migration error occurs. Continuing from this error initializes the variable to nil. If you do not wish to lose data, therefore, you must perform some custom conversions.

Explicit Mapping by Name

The first situation requires providing an explicit mapping from the instance variable names of the older class to the instance variable names of the migration destination. To provide such a customized mapping, override the default mapping

strategy by implementing a class method named `instVarMappingTo:` in your destination class.

For example, suppose that you define the class NewAnimal with three instance variables: *species*, *name*, and *diet*. When instances of Animal migrate to NewAnimal, the value to which *species* ought to be initialized cannot be determined. The value of *name* can be retained, and the value of *diet* ought to be initialized with the value presently held in *favoriteFood*. In that case, the class NewAnimal must define a class method that appears as shown in the example below. However, this example is not portable to future versions of the class.

Example 8.8

```
instVarMappingTo: anotherClass
| result myNames itsNames |
result := super instVarMappingTo: anotherClass.
"Use the default strategy first to properly fill in inst
vars having the same name."
myNames := self instVarNames.
itsNames := anotherClass instVarNames.
(itsNames includesValue: #diet) ifFalse: [
      "No, it uses the old name."
      result
            at: (myNames indexOfValue: #diet)
            put:(itsNames indexOfValue: #favoriteFood)].
^result
```

To implement the method so that it can be reused (with whatever minimal changes are required) in version after version, it might be better to do it as shown in Example 8.9:

Example 8.9

```
instVarMappingTo: anotherClass
"Use the default strategy first to properly fill in inst
vars having the same name."
| result myNames itsNames dietIndex|
result := super instVarMappingTo: anotherClass.
myNames := self instVarNames.
itsNames := anotherClass instVarNames.
dietIndex := myNames indexOfValue: #diet.
dietIndex >0
   ifTrue: [(result at: dietIndex) = 0
   "No, it still needs one"
      ifTrue:[ result at: dietIndex
            put:(itsNames indexOfValue: #favoriteFood)]].
^result
```

Transforming Variable Values

Another kind of customization is required when the format of data changes. For example, suppose that you have a class named Point, which defines two instance variables *x* and *y*. These instance variables define the position of the point in Cartesian two-dimensional coordinate space.

Suppose that you define a class named NewPoint to use polar coordinates. The class has two instance variables named *radius* and *angle*. Obviously the default mapping strategy is not going to be helpful here; migrating an instance of Point to become an instance of NewPoint loses its data—its position—completely. Nor is it correct to map *x* to *radius* and *y* to *angle*. Instead, what is needed is a method that implements the appropriate trigonometric function to transform the point to its appropriate position in polar coordinate space.

In this case, the method to override is `migrateFrom: instVarMap:`, which you implement as an instance method of the class NewPoint. Then, when you request an instance of Point to migrate to an instance of NewPoint, the migration code that calls `migrateFrom: instVarMap:` executes the method in NewPoint instead of in Object.

Example 8.10

```
Object subclass: #oldPoint
   instVarNames: #( #x #y )
   classVars: #()
   poolDictionaries: #()
   inDictionary: UserGlobals
   constraints: #[ #[ #x, Number ],
      #[ #y, Number ] ]
   isInvariant: false

oldPoint compileAccessingMethodsFor: oldPoint.instVarNames

Object subclass: #Point
   instVarNames: #( #radius #angle )
   classVars: #()
   poolDictionaries: #()
   inDictionary: UserGlobals
   constraints: #[ #[ #radius, Number ],
      #[ #angle, Number ] ]
   isInvariant: false

Point compileAccessingMethodsFor: Point.instVarNames

method: Point
migrateFrom: oldPoint instVarMap: aMap
      | x y |
      x := oldPoint x.
      y := oldPoint y.
      radius := ((x*x) + (y*y)) asFloat sqrt.
      angle := (x/y) asFloat arcTan.
      ^self
```

Of course, if you believe there is a chance that you might be migrating instances from a completely separate version of class Point that does not have the instance variables x and y, nor use the Cartesian coordinate system, then it is wise to check for the class of the old instance before you determine which method `migrateFrom: instVarMap:` to use.

For example, you could define a class method `isCartesian` for your old class Point that returns true. Other versions of class Point could define the same method

to return false. (You could even define the method in class Object to return false.) You could then incorporate the test in the above method:

Example 8.11

```
method: Point
migrateFrom: oldPoint instVarMap: aMap
| x y |
oldPoint isCartesian
    ifTrue: [
      x := oldPoint x.
      y := oldPoint y.
      radius := ((x*x) + (y*y)) asFloat sqrt.
      angle := (x/y) asFloat arcTan.
      ^self]
    ifFalse: [^super migrateFrom: oldPoint instVarMap: aMap]
```

Handling Constraint Errors

If a new version of a class redefines the constraint on an instance variable, and the new constraint class is not a subclass of the previous version's constraint class, then GemStone raises an error when you try to migrate an instance of the previous version to the new version. This is true for named instance variables as well as unnamed ones—that is, the elements of a collection.

For example, suppose a manufacturing application defines a class named Part to have an instance variable named *id*, constrained to be an instance of Integer as shown below.

Example 8.12

```
Object subclass: #Part
       instVarNames: #( #id )
       classVars: #()
       classInstVars: #()
       poolDictionaries: #()
       inDictionary: UserGlobals
       constraints: #[ #[ #id, Integer ] ]
       instancesInvariant: false
       newVersionOf: nil
       isModifiable: false
Part compileAccessingMethodsFor: Part.instVarNames
```

We can now create an instance of Part with an id of 12345:

Example 8.13

```
UserGlobals at: #myPart put: Part new.
myPart id: 12345.
```

However, a new vendor comes along who will sell us some parts more cheaply. As luck would have it, however, the new vendor uses letters of the alphabet in the part numbers.

We must now create a new class Part whose *id* instance variable is constrained to be an instance of String instead.

Example 8.14

```
Object subclass: #Part2
       instVarNames: #( #id )
       classVars: #()
       classInstVars: #()
       poolDictionaries: #()
       inDictionary: UserGlobals
       constraints: #[ #[ #id, String ] ]
       instancesInvariant: false
       newVersionOf: Part
       isModifiable: false
method: Part2
id: String
^id := String
%

Part compileAccessingMethodsFor: Part.instVarNames
```

We can now set the migration destination for Part to be Part2:

Example 8.15

```
"set the migration destination"
Part migrateTo: Part2
```

But String is not a subclass of Integer.

If we now try to migrate an instance of Part to Part2, therefore, we will get an error because of these incompatible instance variable constraints:

Example 8.16

```
myPart migrate.
_____________________________________________________
GemStone: Error         Nonfatal
The migration of aPart could not occur because instance
variable #id with value 12345 is not an instance of String
Error Category: [GemStone] Number: 2148 Arg Count: 4
Arg 1: aPart
Arg 2: id
Arg 3: 12345
Arg 4: String
```

If we continue from this error, the value in *id* will become nil—tolerable for one instance of Part, perhaps, but certainly not for a whole factory full! Therefore, we handle this error by aborting the transaction and considering the matter more carefully.

The solution for migration problems generally has two parts:

- Defining a migration method or class method in the new class or version of the class; and
- Pulling the old instances through the new class method.

For Part2, a new class method, `migrateFrom:` *aPart*, creates an instance of the new class and converts the instance variable to the new constraint. Prior to returning the new instance, the OOP for the old object is transferred to the new instance.

Example 8.17

```
category: 'Migrating'
classmethod: Part2
migrateFrom: aPart
"Version 1 of Part constrains the Id to be Integer;"
"Version 2 of Part constrains the Id to String. This method"
"should create an instance of Part2 and convert the old"
"Id value to the new value"
| inst |
inst := Part2 new.
"initialize the instance variable to the new constraint."
inst id: ((aPart id) asString).
"then migrates the OOP for aPart to the new object"
inst become: Part2 .
"return the new object."
^ inst
%
Part2 migrateFrom: myPart.
a Part2
  id              12345
```

Migrating Collection Class Objects

GemStone collection classes treat the elements of their collections as unnamed instance variables. Therefore, changing the element constraint of a collection class has the same hazards: if the new constraint class for the elements of a collection is not a subclass of the previous constraint class, a similar error will be raised. The remedy is the same, with one difference—in this case, you override the method `invalidElementConstraintWhenMigratingInto:` *aBag* `for:` *anObject.*

For example, suppose we want to keep track of all the ID numbers of all the different parts the manufacturer orders from specific vendors. We implement a class called IdBag to keep this information; its elements are constrained to be instances of class Integer. For the same reason that we changed Part, however, we must make a new version, IdBag2, whose elements are now constrained to be instances of String. Migrating an instance of IdBag to IdBag2 will raise an error.

The solution parallels the one for an individual part, as shown in Example 8.18:

Example 8.18

```
IdentityBag subclass: #IdBag
     instVarNames: #()
     classVars: #()
     poolDictionaries: #()
     inDictionary: UserGlobals
     constraints: SmallInteger
     isInvariant: false

IdentityBag subclass: #IdBag2
     instVarNames: #()
     classVars: #()
     classInstVars: #()
     poolDictionaries: #()
     inDictionary: UserGlobals
     constraints: String
     instancesInvariant: false
     newVersionOf: IdBag
     isModifiable: false

method: IdBag
invalidElementConstraintWhenMigratingInto: aBag for: anObject
   "The argument aBag must be an instance of the migration
    destination class. The argument anObject is an element
    contained in the receiver."
(aBag isKindOf: IdBag2)
  ifTrue: [
    "convert the element into an object that can be added to aBag"
     ^ anObject asString
  ]
  ifFalse: [
    "invoke superclass method to raise error"
 super invalidElementConstraintWhenMigratingInto:aBag for:anObject
  ]
%
UserGlobals at: #myIdBag put:
 (IdBag new add: 12; add: 34; add: 56; add: 78; add: 90; yourself).
IdBag migrateTo: IdBag2.
```

We can now migrate instances of IdBag to IdBag2 without encountering the error. When the `migrate` message is sent to an instance of IdBag, instead of raising the migration error we will invoke the new method we've just defined.

Chapter 9

File I/O and Operating System Access

As a GemStone application programmer, you'll seldom need to trouble yourself with the details of operating system file management. Occasionally, however, you might wish to transfer GemStone data to or from a text file on the GemStone object server's host machine. This chapter explains how such tasks can be accomplished.

Accessing Files
describes the protocol provided by class GsFile to open and close files, read their contents, and write to them.

Executing Operating System Commands
describes the protocol provided by class System to spawn a new process on the server's machine to execute operating system commands.

Storing Objects and Exchanging Data
introduces the class PassiveObject—the mechanism GemStone provides for storing the objects that represent your data and exchanging data between GemStone repositories.

Creating and Using Sockets
describes the protocol provided by class GsSocket to create operating system sockets and exchange data between two independent interface processes.

9.1 Accessing Files

The class GsFile provides the protocol to create and access operating system files. This section provides a few examples of the more common operations for text files—for a complete description of the functionality available, including the set of messages for manipulating binary files, see the class GsFile in the *GemStone Kernel Reference*.

Specifying Files

Many of the methods in the class GsFile take as arguments a *file specification*, which is any string that constitutes a legal file specification in the operating system under which GemStone is running. Wild card characters are legal in a file specification if they are legal in the operating system.

Many of the methods in the class GsFile distinguish between files on the client versus the server machine. In this context, the term *client* refers to the machine on which the interface is executing, and the *server* refers to the machine on which the Gem is executing. In the case of a linked interface, the interface and the Gem execute as a single process, so the client machine and the server machine are the same. In the case of an RPC interface, the interface and the Gem are separate processes, and the client machine can be different from the server machine.

Specifying Files Using Environment Variables

If you supply an environment variable instead of a full path when using the methods described in this chapter, how the environment variable is expanded depends upon whether the process is running on the client or the server machine. If you are running a linked interface or you are using methods that create processes on the server, the environment variables accessed by your Smalltalk methods are those defined in the shell under which the Gem process is running. If you are running an RPC interface and using methods that create processes on a separate client machine, the environment variables are instead those defined by the remote user account on the client machine on which the application process is running.

NOTE
If you do not wish to concern yourself with such details, supply full path names and avoid the use of UNIX or DOS environment variables. This allows your application to work uniformly across different environments. See your GemBuilder manual.

The examples in this section use a UNIX path as a file specification.

Creating a File

You can create a new operating system file from Smalltalk using several class methods for GsFile. For example, to create a new file on the client machine and open it for writing, execute an expression of the form:

Example 9.1

```
"Note that mySpec was defined as a variable, you should
use 'pathname' to a local file.
myFile := GsFile openWrite: mySpec.
UserGlobals at: #mySpec put: mySpec;
   at: #myFile put: myFile
%
! must close a file on a real file system
myFile close
%
```

The default is text mode.

NOTE

As a client on a Windows NT system, you need to keep track of whether the type of a file is TXT or BIN, because of how that OS treats the two file types for editing. Opening or writing a BIN file in TXT mode may cause portability problems, because of end-of-line characters are different in the two file types. On Unix systems, end-of-line is treated consistently.

To create a new file on the server machine and open it for writing, execute an expression of the form:

Example 9.2

```
myFile := GsFile openWriteOnServer: mySpec
%

! must close a file on a real file system
myFile close
```

These methods return the instance of GsFile that was created, or nil if an error occurred. Common errors include insufficient permissions to open the file for modification. All GemStone error messages are listed in Appendix B, *"GemStone Error Messages"*.

Opening and Closing a File

GsFile provides a wide variety of protocol to open and close files. For a complete list, see the GemStone Kernel Reference manual.

Table 9.1 GsFile Method Summary

`GsFile openRead: aFile`	Opens a file on the client machine for reading, replacing the existing contents. Returns the instance of GsFile that was created; nil if an error occurred
`GsFile openAppend: aFile`	Opens a file on the client machine for reading, appending the new contents instead of replacing the existing contents. Returns the instance of GsFile that was created; nil if an error occurred.
`GsFile openReadOnServer:`	Opens a file on the server for reading, replacing the existing contents. Returns the instance of GsFile that was created; nil if an error occurred.
`GsFile openAppendOnServer:`	Opens a file on the server for reading, appending the new contents instead of replacing the existing contents. Returns the instance of GsFile that was created; nil if an error occurred.
`GsFile close`	Closes the receiver. Returns the receiver if successful; nil if an error occurred.
`GsFile CloseAll`	Closes all open GsFile instances on the client machine except stdin, stdout, and stderr. Returns the receiver if successful; nil if an error occurred.
`GsFile closeAllOnServer`	Closes all open GsFile instances on the server except stdin, stdout, and stderr. Returns the receiver if successful; nil if an error occurred.

Your operating system sets limits the number of files a process can concurrently access; some systems allow this limit to be changed. Using GemStone classes to open, read or write, and close files does not lift your application's responsibility for closing open files. Make sure you write and close files as soon as possible.

Writing to a File

After you have opened a file for writing, you can add new contents to it in several ways. For example, the instance methods `,` `addAll:`, and `nextPutAll:` all take strings as arguments and write the string to the end of the file specified by the receiver. The method `add:` takes a single character as argument and writes the character to the end of the file. And various methods such as `cr`, `lf`, and `ff` write specific characters to the end of the file—in this case, a carriage return, a line feed, and a form feed character, respectively.

For example, the following code writes the two strings specified to the file *myFile.txt*, separated by end-of-line characters.

Example 9.3

```
myFile := GsFile openWrite: mySpec.
myFile nextPutAll: 'All of us are in the gutter,'.
myFile cr; lf.
myFile nextPutAll: 'but some of us are looking at the
stars.'.
GsFile closeAll.
myFile := GsFile openRead: mySpec.
myFile contents
GsFile closeAll.
```

These methods return the number of bytes that were written to the file, or nil if an error occurs.

Reading From a File

Instances of GsFile can be accessed in many of the same ways as instances of Stream subclasses. Like streams, GsFile instances also include the notion of a position, or pointer into the file. When you first open a file, the pointer is positioned at the beginning of the file. Reading or writing elements of the file ordinarily repositions the pointer as if you were processing elements of a stream.

A variety of methods allow you to read some or all of the contents of a file from within Smalltalk. For example, the end of Example 9.3 returns the entire contents of the specified file and positions the pointer at the end of the file.

Example 9.4 returns the next character after the current pointer position and advances the pointer by one character.

Example 9.4

```
| myString |
myString := String new.
myFile := GsFile openRead: mySpec.
myFile next: 12 into: myString
myFile close
%
```

`Next: into:` returns the twelve characters after the current pointer position and places them into the specified string object. It then advances the pointer by twelve characters.

These methods return nil if an error occurs.

Positioning

You can also reposition the pointer without reading characters, or peek at characters without repositioning the pointer. For example, the following code allows you to view the next character in the file without advancing the pointer.

Example 9.5

```
myFile peek
```

Example 9.6 allows you to advance the pointer by 16 characters without reading the intervening characters.

Example 9.6

```
myFile skip: 16
```

Testing Files

The class GsFile provides a variety of methods that allow you to determine facts about a file. For example, the following code test to see whether the specified file exists on the client machine:

Example 9.7

```
GsFile exists: '/tmp/myfile.txt'
```

This method returns true if the file exists, false if it does not, and nil if an error occurred. To determine if the file exists on the server machine, use the method `existsOnServer:` instead.

To determine if a specified file is open, execute an expression of the form:

Example 9.8

```
myFile isOpen
myFile fileSize
myFile pathName
```

Removing Files

To remove a file from the client machine, use an expression of the form:

Example 9.9

```
GsFile closeAll.
GsFile removeClientFile: mySpec.
%
```

To remove a file from the server machine, use the method `removeServerFile:` instead. These methods return the receiver or nil if an error occurred.

Examining A Directory

To get a list of the names of files in a directory, send GsFile the message `contentsOfDirectory:` *aFileSpec* `onClient:` *aBoolean*. This message acts very much like the UNIX *ls* command, returning an array of file specifications for all entries in the directory.

If the argument to the `onClient:` keyword is true, GemStone searches on the client machine. If the argument is false, it searches on the server instead.

For example:

Example 9.10

```
GsFile  contentsOfDirectory: '/usr/tmp/'  onClient: true
```

If the argument is a directory name, this message returns the full pathnames of all files in the directory, as shown in Example 9.10 above. However, if the argument is a file name, this message returns the full pathnames of all files in the current directory that match the file name. The argument can contain wild card characters such as *. Example 9.11 shows a different use of this message:

Example 9.11

```
GsFile
      contentsOfDirectory: '/tmp/*.c'
      onClient: false
 an Array
      #1 /user2/ateam/personnel/godel.c
      #2 /user2/ateam/personnel/escher.c
      #3 /user2/ateam/personnel/bach.c
```

If you wish to distinguish between files and directories, you can use the message `contentsAndTypesOfDirectory: onClient:` instead. This method returns an array of pairs of elements. After the name of the directory element, a value of true indicates a file; a value of false indicates a directory. For example:

Example 9.12

```
GsFile
      contentsAndTypesOfDirectory: '/tmp/personal/'
      onClient: true
 an Array
      /user2/ateam/personnel/tom    true
      /user2/ateam/personnel/dick   true
      /user2/ateam/personnel/harry  true
      /user2/ateam/personnel/Temps  false
```

All the above methods return nil if an error occurs.

9.2 Executing Operating System Commands

System also understands the message `performOnServer:` *aString*, which causes the UNIX shell commands given in *aString* to execute in a subprocess of the current GemStone process. The output of the commands is returned as a Smalltalk string. For example:

Example 9.13

```
System performOnServer: 'date'
 Sat Jan 1 1:22:56 PST 1994
```

The commands in *aString* can have exactly the same form as a shell script; for example, new lines or semicolons can separate commands, and the character “\” can be used as an escape character. The string returned is whatever an equivalent shell command writes to *stdout*. If the command or commands cannot be executed successfully by the subprocess, the interpreter halts and GemStone returns an error message.

9.3 File In, File Out, and Passive Object

To archive your application or transfer GemStone classes to another repository you can *file out* Smalltalk source code for classes and methods to a text file. To port your application to another repository, you can *file in* that text file, and the source code for your classes and methods is immediately available in the new repository.

Objects representing your data are stored for transfer to another repository with the GemStone class PassiveObject. PassiveObject starts with a root object and traces through its instance variables, and *their* instance variables, recursively until it reaches atomic objects (characters, integers, booleans, or nil), or classes that can be reduced to atomic objects (strings and numbers that are not integers), creating a representation of the object that preserves all of the values required to re-create it. The resulting *network* of object descriptions can be written to a file, stream, or string. Each file can hold only one network—you cannot append additional networks to
an existing passive object file, stream, or string.

A few objects and aspects of objects are not preserved:

- Instances of UserProfile cannot be preserved in this way, for obvious security reasons.
- Instances of Segment and SystemRepository cannot be preserved.
- Blocks that refer to globals or other variables outside the scope of the block cannot be reactivated correctly.
- Segment assignments and blocks that can be associated with objects (such as the sort block in SortedCollections) are not preserved.
- Any indexes you have created on the object are lost as well.

The relationship between two objects is conserved only so long as they are described in the same network. Similarly, if two separate objects A and B both refer to the same third object C, then making A and B passive in two separate operations will result in duplicating the object C, which will be represented in both A's and B's network. Because the resulting network of objects can be quite large anyway, you want to avoid such unnecessary duplication. For this reason, it is usually a good idea to create one collection to hold all the objects you wish to preserve before invoking one of the PassiveObject methods.

The class PassiveObject implements the method `passivate:` *anObject* `toStream:` *aGsFileOrStream* to write objects out to a stream or a file. To write the object bagOfEmployees out to the file *allEmployees.obj* in the current directory, execute an expression of the form shown in Example 9.14.

Example 9.14

```
run
| bagOfEmployees empFile |
UserGlobals at: #bagOfEmployees put: myEmployees;
   at: #empFile put: (GsFile openWrite: 'allEmployees.obj').
%
expectvalue %GsFile
run
PassiveObject passivate: bagOfEmployees toStream: empFile.
empFile close.
%
```

The class PassiveObject implements the method `newOnStream:` *aGsFileOrStream* to read objects from a stream or file into a repository. The method `activate` then restores the object to its previous form.

The following example reads the file *allEmployees.obj* into a GemStone repository:

Example 9.15

```
expectvalue %GsFile
run
empFile := GsFile openRead: 'allEmployees.obj'.
bagOfEmployees := (PassiveObject newOnStream: empFile) activate.
empFile close.
%
```

Examples 9.14 and 9.15 use streams rather than files to actually move the data. This is useful, as streams do not create temporary objects that occupy large amounts of memory before the garbage collector can reclaim their storage.

If you wish to write the contents directly to a file on either the client or the server machine, you can use a method such as the following:

Example 9.16

```
(bagOfEmployees passivate) toClientTextFile: 'allEmployees.obj'
```

You can use the method `toServerTextFile:` to specify a file on the server machine instead. The passive object can be read into another repository with an expression like the one in Example 9.17:

Example 9.17

```
((PassiveObject new) fromServerTextFile: 'allEmployees.obj')activate
```

Expressions of the form in Examples 9.16 and 9.17 allow you to specify files on specific machines, but they have the disadvantage of creating large temporary objects which occupy inconvenient amounts of storage until the garbage collector reclaims it.

A third strategy allows you to save passive objects in strings that can then be sent through a socket. To do so, use an expression of the form:

Example 9.18

```
expectvalue %SetOfEmployees
run
|theString|
theString := bagOfEmployees passivate contents.
theString toClientTextFile: 'allEmployees.obj'.
((PassiveObject newWithContents: theString)
   fromClientTextFile: 'allEmployees.obj') activate
%
```

9.4 Creating and Using Sockets

Sockets open a connection between two processes, allowing a two-way exchange of data. The class GsSocket provides a mechanism for manipulating operating system sockets from within Smalltalk.

Methods in the class GsSocket do not use the terms *client* and *server* in the same way as the methods in class GsFile. Instead, these terms refer to the roles that two processes play with respect to the socket: the server process creates the socket, binds it to a port number, and listens for the client, while the client connects to an already created socket. Both client and server are processes created (or spawned) by a Gem process.

In the Class definition for GsSocket, under the Examples category, are two examples you can use to create a GsSocket. The example methods work together; they require two separate sessions running from two independently executing interfaces, one running the server example and one running the client example. You can copy the methods to your GemBuilder workspace, change any machine names as necessary, and run them in your workspace. As written, the two processes — server and client — run on the same host machine. However, assuming that two machines are already networked together properly, you need only replace the string `'localhost'` in the `clientExample` method with another string specifying a remote host to run the client example on a different machine from the server example.

The examples create a socket, establish a connection between them, exchange data using instances of PassiveObject, and then close the socket.

NOTE
The method `serverExample` *will take control of the interface that invokes it, allowing no further user input until the socket it creates succeeds in connecting to the client socket. If this happens you need to interrupt the command.*

To run this example, execute the expression `GsSocket serverExample` from one interface before invoking the expression `GsSocket clientExample` from the other interface.

Chapter

10 Signals and Notifiers

This chapter discusses how to communicate between one session and another, and between application and another.

Communicating Between Sessions
introduces two ways to communicate between sessions.

Object Change Notification
describes the process used to enable object change notification for your session.

Gem-to-Gem Signaling
describes one way to pass signals from one session to another.

Performance Considerations
describes areas of concern for developers tuning the performance of their applications.

10.1 Communicating Between Sessions

Applications that handle multiple sessions often find it convenient to allow one session to know about other sessions' activities. GemStone provides two ways to send information from one current session to another:

- *Object change notification*

  Reports the changes recorded by the object server. You set your session to be notified when specific objects are modified. Once enabled, notification is automatic, but a signal is not sent until the changed objects are committed.

- *Gem-to-Gem signaling*

  Reports events that happen outside the transaction space. Currently logged in users signal to send messages to each other. Gems can also pass information that is not necessarily visible to users, such as the name of a queue that needs servicing. Sending a signal requires a specific action by the other Gem; it happens immediately.

Object change notification and Gem-to-Gem signals only reach logged in sessions. For applications that need to track processes continuously, you can create a Gem that runs independently of the user sessions and monitors the system. See the instructions on creating a custom Gem in the *GemBuilder for C* manual.

10.2 Object Change Notification

Object change notifiers are signals that can be generated by the object server to inform you when specified objects have changed. You can request that the object server inform you of these changes by adding objects to your *notify set*.

When a reference to an object is placed in a notify set, you receive notification of all changes to that object (including the changes you commit) until you remove it from your notify set or end your GemStone session. The notification you receive can vary in form and content, depending on which interface to GemStone you are running and how the notification action was defined.

Your application can respond in several ways:

- Prompt users to abort or commit for an updated image
- Log the information in an object change report.
- Use the notifiers to trigger another action. For example, a package for managing investment portfolios might check the stock that triggered the notifier and enter a transaction to buy or sell if the price went below or above preset values.

To set up a simple notifier for an object:

1. Create the object and commit it to the object server.
2. Add the object to your session's notify set with the messages:

   `System addToNotifySet:` *aCommittedObject*
   `System addAllToNotifySet:` *aCollectionofCommittedObjects*

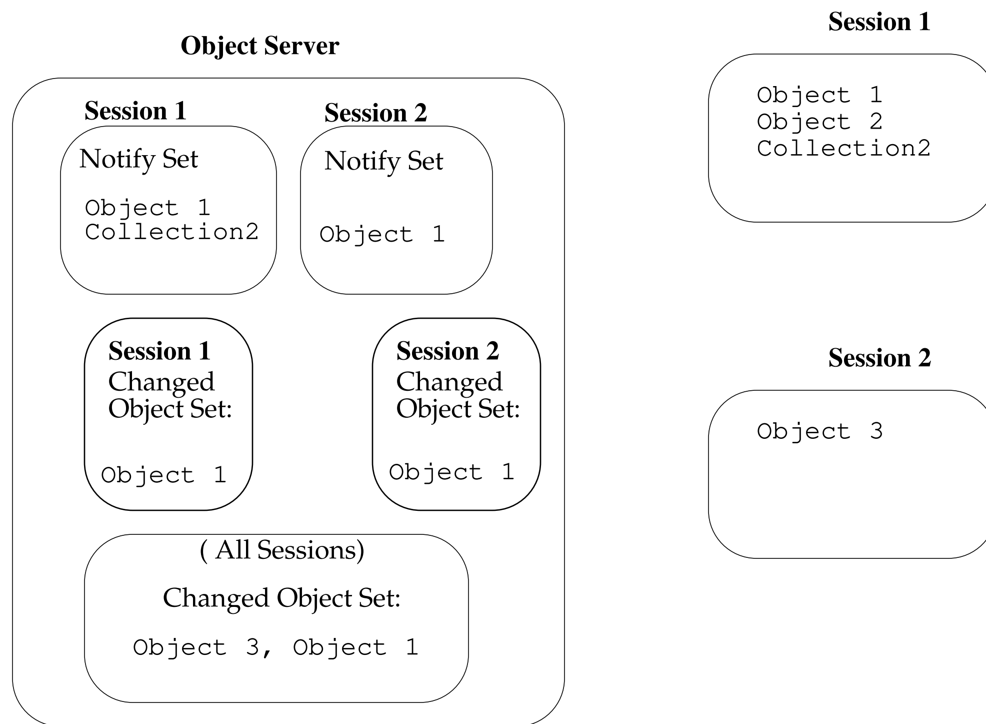
3. Define how to receive the notifier with either a notifier message or by polling.
4. Define what your session will do upon receiving the notifier.

Before discussing each of these steps in detail, the following section outlines the object server's role in monitoring committed objects.

How the Object Server Notifies a Session

After a commit, each session view is updated. The object server also updates its list of committed objects. This list of objects is compared with the contents of the notify set for each session, and a set of the changed objects for each notify set is compiled.

Figure 10.1 The Object Server Tracks Object Changes



In Figure 10.1, both Session 1 and Session 2 want to track changes to Object 1. When Object 1 is committed to the repository, the object server checks the notify sets for Session 1 and Session 2. The object server then sends the signal `#rtErrSignalCommit`. If your session has enabled this signal, your session receives this signal. Upon receiving this signal, you choose what action to take.

At this point, your session only knows that some object you are interested in has changed. To find out which objects in your notify set have changed, you need to ask the object server to send you the changed object set. If you choose not to enable the signal, you can ask the object server for your session's set of changed objects at any time.

To receive the set of changed objects, your session sends `System | signaledObjects`. This method returns the OOP for objects in the signaled object set, then clears the signaled object set.

Your session can now use an exception handler to perform some action concerning the specific objects that changed.

Setting Up a Notify Set

GemStone supplies a notify set for each user session to which you add or remove objects. Except for a few special cases discussed later, any object you can refer to can be added to a notify set.

Notify sets persist through transactions, living as long as the GemStone session in which they were created. When the session ends, the notify set is no longer in effect. If you need notification regarding the same objects for your next session, you must once again add those objects to the notify set.

Adding Objects to a Notify Set

You can add an object to your notify set with an expression of the form:

`System addToNotifySet:` *aCommittedObject*

When you add an object to the notify set, GemStone begins monitoring changes to it immediately.

Most GemStone objects are composite objects, made up of a root object and a few subobjects. Usually you can just ignore the subobjects. However, if notification is not working on a composite object in your notify set, look at the code and see which objects are actually modified during common operations such as `add:` or `remove:`. For some classes, both the root object and the subobjects must appear in the notify set.

Example 10.1 creates a dictionary of stock holdings and then creates a notify set for the stocks in the holding. Lastly, the session is set to automatically receive the notifier.

Example 10.1 Adding a New Object to the Notify Set

```
" Create a Class to record stock name, number and price: "
(Object subclass: #Holding
  instVarNames: #('name' 'number' 'price')
  classVars: #()
  poolDictionaries: #[]
  inDictionary: PublicDictionary
  constraints: #[]
  instancesInvariant: false
  isModifiable: false) name
%
Holding compileAccessingMethodsFor: Holding instVarNames
" Add an Instance of Holding to the UserGlobals dictionary
  as a StringKeyValue Dictionary: "
UserGlobals
   at: #MyHoldings put: StringKeyValueDictionary new
! Add some stocks to my dictionary:
MyHoldings at: #USSteel put:
   (Holding new name: #USSteel;
      number: 100000; price: 120.00).
MyHoldings at: #SallieMae put:
   (Holding new name: #SallieMae;
      number: 1000; price: 95.00).
%
MyHoldings at: #ATT put:
   (Holding new name: #ATT;
      number: 100000; price: 150.00).
%
commit
"Add the collection object to the notify set"
System addToNotifySet: MyHoldings.
(System notifySet) includesIdentical: MyHoldings
%
run
System enableSignaledObjectsError.
%
```

Objects That Cannot Be Added

Not every object can be added to a notify set. Objects in a notify set must be visible to more than one session; otherwise, other sessions could not change them. So, objects you have created for temporary use or have not committed cannot be added to a notify set. GemStone responds with an error if you try to add such objects to the notify set.

You also receive an error if you attempt to add objects whose values cannot be changed. This includes atomic objects such as instances of Character, SmallInteger, Boolean, or nil.

Collections

You can add a collection of objects to your notify set with an expression like this:

`System addAllToNotifySet:` *aCollectionOfCommittedObjects*

This expression adds the elements of the collection to the notify set.

You don't have to add the collection object itself, but if you do, use `addToNotifySet:` rather than `addAllToNotifySet:`. When a collection object is in the notify set, adding elements to the collection or removing elements from it trigger notification. Modifications to the elements do not trigger notification on the collection object; add the elements to the notification set if you want to know when they change.

Example 10.2 shows the notify set containing both the collection object and the elements in the collection.

Example 10.2

```
| notifyObjs |
"Add the stocks in the collection to the notify set"
System addAllToNotifySet: MyHoldings.
notifyObjs := System notifySet.
notifyObjs includesIdentical: (MyHoldings at: #SallieMae);
   includesIdentical: (MyHoldings at: #ATT);
   includesIdentical: (MyHoldings at: #USSteel)
%
"Add the collection object to the notify set"
System addToNotifySet: MyHoldings.
(System notifySet) includesIdentical: MyHoldings
%
```

Very Large Notify Sets

You can register any number of objects for notification, but very large notify sets can degrade system performance. GemStone can handle up to 2,000 objects for a single session or a total of 5,000 objects across all sessions without significant impact. Beyond that, test whether the response times are acceptable for your application.

If performance is a problem, you can set up a more formal system of change recording. Have each session maintain its own list of the last several objects updated. The list is a collection written only by that session. Create a global collection of collections that contains every session's list of changes. Put the global collection and its elements in your modify set, so you receive notification when a session commits a modified list of changed objects. Then you can check for changes of interest. Keeping a global collection of changes in your modify set preserves the order of the additions, so the new objects can be serviced in the correct order. Notification on a batch of changed objects is received in OOP order.

Listing Your Notify Set

To determine the objects in your notify set, execute:

```
System notifySet
```

Removing Objects From Your Notify Set

To remove an object from your notify set, use an expression of the form:

`System removeFromNotifySet:` *anObject*

To remove a collection of objects from your notify set, use an expression of the form:

`System removeAllFromNotifySet:` *aCollection*

This exprtession removes the elements of the collection. If the collection object itself is also in the notify set, remove it separately, using `removeFromNotifySet:`.

To remove all objects from your notify set, execute:

```
System clearNotifySet
```

NOTE:
D not clear your notify set after each transaction. If you clear the notify set, and then add some of the same objects to it again, you may miss intermediate changes to those objects.

Notification of New Objects

In a multi-user environment, objects are created in various sessions, committed, and immediately open to modification. To receive notifiers on the objects that existed at the beginning of your session may not be enough. You may also need notification concerning new objects.

You cannot put unknown objects in your notify set, but you can create a collection for those kinds of objects and add that collection to the notify set. Then when the collection changes, meaning that objects have been added or removed, you can stop and look for new objects. For example, suppose you want notification when the price of any stock in your portfolio changes. Use the following steps:

1. Create a globally known collection (for example, MyHoldings) and add your existing stock holdings (instances of class Holding) to it.
2. Place these stocks in your notify set:
   ```
   System addAllToNotifySet: MyHoldings
   ```
3. Place MyHoldings in your notify set, so you receive notification that the collection has changed when a stock is bought or sold:
   ```
   System addToNotifySet: MyHoldings
   ```
4. Place new stock purchases in MyHoldings by adding code to the instance creation method for class Holding.
5. When you receive notification that MyHoldings has been changed, compare the new MyHoldings with the original.
6. When you find new stocks, add them to your notify set, so you will be notified if they are changed.

Example 10.3 shows one way to do steps 5 and 6.

Example 10.3 Adding New Objects to Another Session's Notify Set

```
"Make a temporary copy of the set."

| tmp newObjs |
tmp := MyHoldings copy.
"Refresh the view (commit or abort)."
System commitTransaction.
"Get the difference between the old and new sets."
newObjs := (MyHoldings - tmp).
"Add the new elements to the notify set."
newObjs size > 0 ifTrue: [System addAllToNotifySet: newObjs].
```

You can also identify objects to remove from the notify set by doing the opposite operation:

```
tmp - MyHoldings
```

This method could be useful if you are tracking a great many objects and trying to keep the notify set as small as possible.

Receiving Object Change Notification

You can receive notification of committed changes to the objects in your notify set in two ways:

- Enabling automatic notification, which is faster and uses less CPU
- Polling for changes

Automatic Notification of Object Changes

For automatic notification, you enable your session to receive the event signal #rtErrSignalCommit. By default, #rtErrSignalCommit is disabled (except in GemBuilder for Smalltalk, which enables the signal as part of `GSSession | notificationAction:`).

To enable the event signal for your session, execute:

```
System enableSignaledObjectsError
```

To disable the event signal, send the message:

```
System disableSignaledObjectsError
```

To determine whether this error message is enabled or disabled for your session, send the message:

```
System signaledObjectsErrorStatus
```

This method returns true if the signal is enabled, and false if it is disabled.

This setting is not affected by commits or aborts. It remains until you change it, you end the session, or you receive the signal. The signal is automatically disabled when you receive it so the exception handler can take appropriate action.

The receiving session traps the signal with an exception handler. Your exception handler is responsible for reading the set of signaled objects (by sending the message `System signaledObjects`) as well as taking the appropriate action.

System | signaledObjects

The `System |signaledObjects` method reads the incoming changed object signals. This method returns an array. The array includes all the objects in your notify set that have changed since the last time you sent `signaledObjects` in your current session. The array contains objects changed and committed by all sessions, including your own. If more than one session has committed, the OOPs are OR'd together. The elements of the array are arranged in OOP order, not in the order the changes were committed. If none of the objects in your notify set have been changed, the array is empty.

NOTE:
Objects that participate in indexes are considered changed if an index is created or removed. You may therefore sometimes receive notification of changes that are normally invisible to you and irrelevant to your purposes.

Use a loop to call `signaledObjects` repeatedly, until it returns a null. The nullguarantees that there are no more signals in the queue.

NOTE:
If users are frequently committing many changes to the objects in your notify set, you may not receive notification for each change. You may not be able to poll frequently enough, or your exception handler may not be able to process the errors it receives fast enough. In such cases, you may miss intermediate values of frequently changing objects.

Polling for Changes to Objects

You also use `System | signaledObjects` to poll for changes to objects in your notify set.

Example 10.4 uses the polling method to inform you if anyone has added objects to a set or changed an existing one. Notice that the set is created in a dictionary that is accessible to other users, not in UserGlobals.

Example 10.4

```
System disableSignaledObjectsError;
   signaledObjectsErrorStatus
  "Create a set."
UserGlobals at: #Changes put: IdentitySet new.
System commitTransaction

System addToNotifySet: Changes
%
Changes add: 'here is a change'.
System commitTransaction
| newSymbols count |
System abortTransaction.
count := 0 .
[ newSymbols := System signaledObjects.
  newSymbols size = 0 and:[ count < 50]
]
  whileTrue: [
   System sleep: 10 .
   count := count + 1
   ].
^ newSymbols.
%
Changes segment worldAuthorization: #write.
System commitTransaction
```

Troubleshooting

Notification on object changes may occasionally produce unexpected results. The following sections outline areas of concern:

Indexes

Objects that participate in indexes are considered changed if an index is created or removed. You may, therefore, sometimes receive notification of changes that are normally invisible to you and irrelevant to your purposes.

Frequently Changing Objects

If users are committing many changes to objects in your notify set, you may not receive notification of each change. You might not be able to poll frequently enough, or your exception handler might not process the errors it receives fast enough. In such cases, you can miss some intermediate values of frequently changing objects.

Special Classes

Most GemStone objects are composite objects, but for the purposes of notification you can usually ignore this fact. They are almost always implemented so that changes to subobjects affect the root, so only the root object needs to go into the notify set. For example, these common operations trigger notification on the root object:

Example 10.5

```
!common operations that trigger notifers

"assignment to an instance variable"
name := 'dowJones'
"updating the indexable portion of an object"
self at: 3 put: 'active'
"adding to a collection"
self add: 3
```

In a few cases, however, the changes are made only to subobjects. For the following GemStone kernel classes, both the object and the subobjects must appear in the notification set:

- RcQueue
- RcIdentityBag
- RcCounter
- KeyValueDictionary.

You can also have the problem with your own application classes. Wherever possible, implement objects so that changes modify the root object, although you also have to balance the needs of notification with potential problems of concurrency conflicts.

If you are not being notified of changes to a composite object in your notify set, look at the code and see which objects are actually modified during common operations such as `add:` or `remove:`. When you are looking for the code that actually modifies an object, you may have to check a lower-level method to find where the work is performed.

Once you know the object's structure and have discovered which elements are changed, add the object and its relevant elements to the notify set. For cases where elements are known, you can add them just like any other object:

`System addToNotifySet:` *anObject*

Example 10.6 shows a method that creates an object and automatically adds it to the notify set in the process.

Example 10.6

```
!Adding Subobjects to the Notify Set
method: SetOfHoldings
add: anObject
      System addToNotifySet: anObject.
      ^super add: anObject
%
```

Methods for Object Notification

Methods related to notification are implemented in class System. Browse the class System and read about these methods.

```
addAllToNotifySet:
addToNotifySet:
clearNotifySet
removeAllFromNotifySet:
removeFromNotifySet:
notifySet
signaledObjects
enableSignaledObjectsError
disableSignaledObjectsError
signaledObjectsErrorStatus
```

Class Exception provides the behavior for capturing signals. Look at these methods related to exception handling:

```
installStaticException: category: number: subtype:
removeStaticException:
category: number: do:
remove
```

10.3 Gem-to-Gem Signaling

GemStone enables you to send a signal from your Gem session to any other current Gem session. GsSession implements several methods for communicating between two sessions. Unlike object change notification, inter-session signaling operates on the event layer and deals with events that are not being recorded in the repository. Signaling happens immediately, without waiting for a commit.

An application can use signals between sessions for situations like a queue, when you want to pass the information quickly. Signals can also be a way for one user who is currently logged in to send information to another user who is logged in.

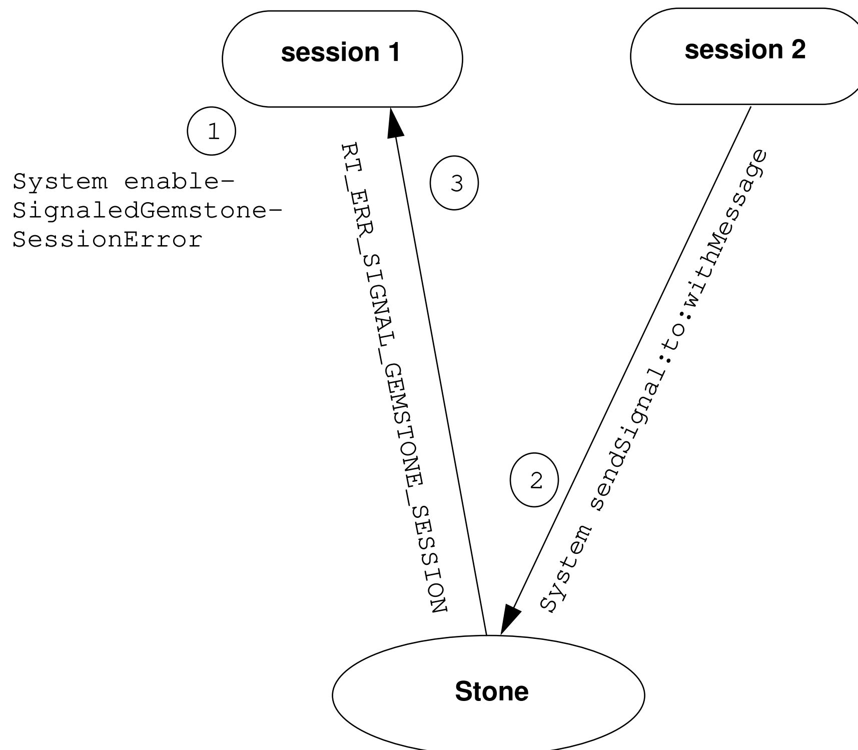
NOTE:
A signal is not exactly an interrupt, and it does not automatically awaken an idle session. The signal can be received only when your GemBuilder interface is active.

You can receive a signal from another session by polling for the signal or by receiving automatic notification.

When the signal is received by polling, the session sends out the message `System signalFromGemStoneSession` at regular intervals.

The receiving session processes the signal with an exception handler. Figure 10.2 shows the type of Gem-to-Gem signaling where the Stone sends an error to the receiving session. The receiving session processes the signal with an exception handler.

Figure 10.2 Communicating from Session to Session



Class Exception provides the behavior for capturing signals. Look at these methods related to exception handling:

```
installStaticException: category: number: subtype:
removeStaticException:
category: number: do:
remove
```

Sending a Signal

To communicate, one session must send a signal and the receiving session must be set up to receive the signal.

Finding the Session ID

To send a signal to another Gem session, you must know its session identification number. To see a description of currently logged in sessions, execute the following method:

```
System currentSessions
```

This message returns an array of SmallInteger representing session identification numbers for all current sessions. Example 10.7 uses this method to find the session ID for user1 and send a message.

Example 10.7

```
| sessionId serialNum aGsSession otherSession signalToSend|
  sessionId := System currentSessions
   detect:[:each |(((System descriptionOfSession: each)
at: 1)
   userId = 'user1') ]
   ifNone: [nil].
sessionId notNil ifTrue: [
  serialNum := GsSession serialOfSession: sessionId .
  otherSession := GsSession sessionWithSerialNumber:
serialNum .
  signalToSend :=
    GsInterSessionSignal signal: 4
                         message:'reinvest form is here'.
"one of two ways to send it"
  signalToSend sendToSession: otherSession .
"ALTERNATIVE CODE:
     otherSession sendSignalObject: signalToSend ."
  ]
```

The other session needs to receive the message, as shown in this example:

Example 10.8

```
GsSession currentSession signalFromSession message

reinvest form is here
```

Sending the Message

When you have the session identification number, use the method `GsInterSessionSignal | signal: aSignalNumber message: aMessage`.

aSignalNumber is determined by the particular protocol you arranged at your site and the specific message you wish to send. Sending the integer "1," for example, doesn't convey a lot unless everyone has agreed that "1" means "Ready to trade." You could set up an application-level symbol dictionary of meanings for the different signal numbers, similar to the standard GemStone error dictionary discussed in "Signaling Errors to the User" on page 11-1.

aMessage is a String object with up to 1023 characters.

Instead of assigning meanings to aSignalNumber, your site might agree that the integer is meaningless, but the message string is to be read as a string of characters conveying the intended message, as in Example 10.9.

For more complex information, the message could be a code where each symbol conveys its own meaning.

You can use signals to broadcast a message to every user logged in to GemStone. In Example 10.9, one session notifies all current sessions that it has created a new object to represent a stock that was added to the portfolio. In applications that commit whenever a new object is created, this code could be part of the instance creation method for class Holding. Otherwise, it could be application-level code, triggered by a commit.

Example 10.9

```
System currentSessions do: [:each |
        System sendSignal: 8 to: each
               withMessage: 'new Holding: SallieMae'.]
```

If the message is displayed to users, they can commit or abort to get a new view of the repository and put the new object in their notify sets. Or the application could be set up so that signal 8 is handled without user visibility. The application might do an automatic abort, or automatically start a transaction if the user is not in one, and add the object to the notify set. This enables setting up a notifier on a new unknown object. Also, because signals are queued in the order received, you can service them in order.

Receiving a Signal

You can receive a signal from another session in either of two ways: you can poll for such signals, or you can enable a signal from GemStone. Signals are queued in the receiving session in the order in which they were received. If the receiving session has inadequate heap space for an incoming signal, the contents of the signal is written to *stdout*, whether the receiving session has enabled receiving such signals or not. (Both the structure of the signal contents and the process of enabling signals are described in detail in the next sections.)

The `System | signalFromGemStoneSession` method reads the incoming signals, whether you poll or receive a signal. If there are no pending signals, the array is empty.

Use a loop to call `signalFromGemStoneSession` repeatedly, until it returns a null. This guarantees that there are no more signals in the queue. If signals are being sent quickly, you may not receive a separate RT_ERR_SIGNAL_GEMSTONE_SESSION for every signal. Or, if you use polling, signals may arrive more often than your polling frequency.

Polling

To poll for signals from other sessions, send the following message as often as you require:

```
System signalFromGemStoneSession
```

If a signal has been sent, this method returns a three-element array containing:

- the session identification number of the session that sent the signal (a SmallInteger),
- the signal value (a SmallInteger), and
- the string containing the signal message.

If no signal has been sent, this method returns an empty array.

Example 10.10 shows how to poll for Gem-to-Gem signals. If the polling process finds a signal, it immediately checks for another one until the queue is empty. Then the process sleeps for 10 seconds.

Example 10.10

```
run
| response |
[ response := System signalFromGemStoneSession.
  response size = 0 ] whileTrue:[System sleep: 10].
^response
%
```

Receiving an Error Message from the Stone

To use the error mechanism to receive signals from other Gem sessions, you must enable the error #rtErrSignalGemStoneSession. This error has three arguments:

- the session identification number of the session that sent the signal (a SmallInteger)
- the signal value (a SmallInteger)
- the string containing the signal message

By default, the error #rtErrSignalGemStoneSession is disabled, except in the GSI interface, which enables the error as part of `GSSession | gemSignalAction:`.

To enable this error, execute:

```
System enableSignaledGemStoneSessionError
```

To disable the error, send the message:

```
System disableSignaledGemStoneSessionError
```

To determine whether receiving this error message is presently enabled or disabled, send the message:

```
System signaledGemStoneSessionErrorStatus
```

This method returns true if the error is enabled, and false if it is disabled.

This setting is not affected by commits or aborts. It remains until you change it, you end the session, or you receive the error. The error is automatically disabled when you receive it so the exception handler can take appropriate action without further interruption.

10.4 Performance Considerations

GemStone notifiers and Gem-to-Gem signals use the same underlying implementation. You can use the suggestions in this section to improve the performance of applications that depend on either mechanism.

Increasing Speed

Signals and notifiers require a few milliseconds to get to their destination, no matter how you set up your application. You can improve the speed by using linked Gems, rather than separate RPC sessions.

Receiving the signal can also be delayed. GemStone is not an interrupt-driven application programming interface. It is designed to make no demands on the application until the application specifically requests service. Therefore, Gem-to-Gem signals and object change notifiers are not implemented as interrupts, and they do not automatically awaken an idle session. They can only be received when GemBuilder is running, not when you are running client code, sitting at the Topaz prompt, writing to a socket connection, or waiting for a child process to complete. The signals are queued up and wait until you read them, which can create a problem with signal overflow if the delay is too long and the signals are coming rapidly.

You can receive signals at reliable intervals by regularly performing some operation that activates Gembuilder. For example, in a Smalltalk application you could set up a polling process that periodically sends out `GSSession | pollForSignal`. The `pollForSignal` method causes GemBuilder to poll the repository. GemBuilder for C also provides a wrapper for the function **GciPollForSignal**.

You should also check in your application to make sure the session does not hang. For instance, use `GsSocket | readReady` to make sure your session won't be waiting for nonexistent input at a socket connection.

See "Using Signals and Notifiers with RPC Applications" on page 10-23.

Dealing With Signal Overflow

Gem-to-Gem signals and object change notification signals are queued separately in the receiving session. The queues maintain the order in which the signals are received.

NOTE:
For object change notification, the queue does not preserve the order in which the changes were committed to the repository. Each notification signal contains an array of OOPs, and these changes are arranged in OOP order. See "Receiving Object Change Notification" on page 10-10.

If the receiving session has inadequate heap space for an incoming signal, the contents of the signal are written to *stdout* whether the receiving session has enabled such signals or not. The internal buffer for Gem-to-Gem signals is approximately 9 KBytes per receiving session. This 9 KBytes must accommodate any strings sent as part of the signal, along with some other things like message headers. If the receiving side is completely idle, 9 KBytes is the maximum size of signal data that another session could send. If the receiving side *is* receiving, there is no hard limit, but if the sending side is running slightly faster, then eventually the sender will overflow the buffer.

When the output buffer from the Stone to the target session is full, error 2254, #errSesBlockedOnOutput, is raised. Set your application so the sender gracefully handles this error. For example, the sender might try to send the signal five times, and finally display a message of the form:

```
Receiver not responding.
```

The most effective way to prevent signal overflow is to keep the session in a state to receive signals regularly, using the techniques discussed in the preceding section. When you do receive signals, make sure you read all the signals off the

queue. Repeat `signaledObjects` or `signalFromGemStoneSession` until it returns a null. You can postpone the problem by sending very short messages, such as an OOP pointing to some string on disk or perhaps an index into a global message table. Check `System | sendSignal: to: withMessage:` for a better idea of how the message queue works.

Using Signals and Notifiers with RPC Applications

RPC user applications need to call **GciPollForSignal** regularly to receive the signal from the Gem. For linked applications, this call is not necessary, because the applications run as part of the same process as the Gem. See your GemBuilder interface manual for more information.

Sending Large Amounts of Data

If you want to pass large amounts of data between sessions, sockets are more appropriate than Gem-to-Gem signals. Chapter 9, "File I/O and Operating System Access" describes the GemStone interface to TCP/IP sockets. That solution does not pass data through the Stone, so it does not create system overload when you send a great many messages or very long ones.

Maintaining Signals and Notification When Users Log Out

Object change notification and Gem-to-Gem signals only reach logged in sessions. For applications that need to track processes continuously, you can create a Gem that runs independently of the user sessions and monitors the system. In manufacturing, such a Gem can monitor a machine and send a warning to all current sessions when something is out of tolerance. Or it might receive the information that all the users need and store it where they can find it when they log in.

Example 10.11 shows some of the code executed by an error handler installed in a monitor Gem. It traps Gem-to-Gem signals and writes them to a log file.

Example 10.11 Logging Gem-to-Gem Signals

```
run
| gemMessage logString |
gemMessage := System signalFromGemStoneSession.
logString := String new.
logString add:
'-----------------------------------------------------
The signal ';
      add: (gemMessage at: 2) asString;
      add: ' was received from GemStone sessionId = ';
      add: (gemMessage at: 1) asString;
      add: ' and the message is ';
      addAll: (gemMessage at: 3).
logString toServerTextFile: ’user2/trading/logdir’ +
                              '/gemmessage.txt'.
%
```

Chapter 11

Error Handling

GemStone provides several mechanisms that allow you to deal with errors in your programs.

Signaling Errors to the User
describes the mechanism whereby an application can halt execution and report errors to the user.

Handling Errors in Your Application
describes the class Exception, which allows you to define categories of errors and install handlers in your application to cope with them without halting execution.

11.1 Signaling Errors to the User

Class System provides a facility to help you trap and report errors in your Smalltalkprograms. When you send a message of the form:

`System signal:` *anInt* `args:` *anArray* `signalDictionary:` *aDict*

System looks up an object identified by the number *anInt* in the SymbolDictionary *aDict*. Using that object and any information you included in *anArray*, it builds a

string that it passes back to the user interface code as an error description. The Smalltalk interpreter halts.

Suppose, for example, that you create a SymbolDictionary called MyErrors in which the string 'Employee age out of range' is identified by the number 1. The following method causes that string to be passed back to the user interface whenever the method's argument is out of range.

Example 11.1

```
method: Employee
age: anInt
(anInt between: 15 and: 65)
    ifFalse: [System signal: 1 args: #() signalDictionary:
MyErrors].
age := anInt.
%
```

The SymbolDictionary containing error information is actually keyed on symbols such as #English or #Kwakiutl that name natural languages. Each key is associated with an array of error-describing objects. Here, in Example 11.2, is a SymbolDictionary containing English and Pig Latin error descriptions:

Example 11.2

```
| signalDict |
signalDict := SymbolDictionary new.
signalDict at: #English put: Array new;
           at: #PigLatin put: Array new.
(signalDict at: #English)
   at: 1 put: #('Employee age out of range');
   at: 2 put: #('Distasteful input').
(signalDict at: #PigLatin)
   at: 1 put: #('Employeeay ageay outay ofay angeray');
   at: 2 put: #('Istastefulday inputay').
UserGlobals at: #MyErrors put: signalDict.
%
```

The error string to be returned in response to a particular signal number depends on the value of instance variable *nativeLanguage* in your UserProfile. The message `nativeLanguage` lets you read the value of that variable, and the message

`nativeLanguage:` lets you change it. Assuming you have the necessary authorization, the following code causes GemStone to respond to you in Pig Latin:

```
System myUserProfile nativeLanguage: #PigLatin
```

If you define the method `Employee | age:` as shown above, then this expression:

```
myEmployee age: -1
```

elicits the error report 'Employeeay ageay outay ofay angeray'.

NOTE:
Signal 0 (zero) is reserved for use by GemStone. Do not use it.

As the previous examples have shown, each error object is an array. Although the arrays in the previous example contained only strings, they can also include SmallIntegers that act as indexes into the parameter to `args:`. When the error string is constructed, each positive SmallInteger in the error object is replaced by the result of sending `asString` to the corresponding element of the `args:` array. This lets you capture and report some diagnostic information from the context of the error.

Suppose, for example, that you wanted to report the actual argument to `age:` that triggered the "out of range" error. You can define your error dictionary this way:

Example 11.3

```
| signalDict |
signalDict := SymbolDictionary new.
signalDict at: #English put: Array new;
           at: #PigLatin put: Array new.
(signalDict at: #English)
   at: 1 put: #('Employee age ' 1 ' out of range');
   at: 2 put: #('Distasteful input').
(signalDict at: #PigLatin)
   at: 1 put: #('Employeeay ageay ' 1 ' outay ofay
angeray');
   at: 2 put: #('Istastefulday inputay').
UserGlobals at: #MyErrors put: signalDict.
```

and then define `age:` like this:

Example 11.4

```
method: Employee
age: anInt
(anInt between: 15 and: 65)
   ifFalse: [System signal: 1 args: #[anInt]
signalDictionary: MyErrors].
age := anInt.
%
```

When an argument to `age:` is out of range, System looks up the array representing the error for signal 1 and begins building a string. The first part of that string is 'Employee age' (the first element of the array), the second part is the result of sending `asString` to *anInt*, and the final part is ' out of range' (the third element of the array). The resulting string has the form 'Employee age –1 out of range'.

The following examples show how you can use a two-element argument array:

Example 11.5

```
(MyErrors at: #English)
  at: 1 put: #('The employee named ' 2 ' cannot be ' 1 ' years
old');
  at: 2 put: #('Distasteful input'). . . .
method: Employee
age: anInt
   (anInt between: 15 and: 65)
      ifFalse: [System signal: 1 args: #[anInt, self name]
                                signalDictionary: MyErrors].
   age := anInt.
%
```

This mechanism's use of `asString` in building error reports does not work if the error involves an authorization problem. For example, then sending `asString` to the object for which you have no authorization only compounds the problem. To help you circumvent this, GemStone also gives you the means to report an object's identifier.

If one of the SmallIntegers in the error object is negative, the absolute value of that number is used for indexing into the `args:` array. Then, when the error string is constructed, the negative SmallInteger is replaced by the identifier of the

corresponding object. For example, if the error object contains the SmallInteger -3, then the error string contains the identifier of the third element of the `args:` array. Because GemStone can report an object's identifier even if you have no read authorization for the object, the error mechanism can't be halted by an authorization error.

In the following examples, the error-handling code given earlier is modified to report the identifier of any employee receiving the message `age:` with an inappropriate argument.

Example 11.6

```
(MyErrors at: #English)
  at: 1 put: #('The employee with object identifier ' -2
              ' cannot be ' 1 ' years old');
  at: 2 put: #('Distasteful input').
method: Employee
age: anInt
   (anInt between: 15 and: 65)
          ifFalse: [System signal: 1 args: #[anInt, self]
                                    signalDictionary: MyErrors].
   age := anInt.
%
```

Invoking `age:` with an out-of-range argument now elicits an error report of the form "The employee with object identifier 77946 cannot be –1 years old."

11.2 Handling Errors in Your Application

Unless an error is fatal to GemStone, it can be handled in your application without halting execution with the class Exception. You define a category of errors to which your application must respond, raise the error under appropriate circumstances, and execute additional Smalltalkcode to recover from the error gracefully.

If no GemStone Smalltalk exception handler is defined for a given error, control returns to the interface you are using. For example, in the GemBuilder for C interface, you handle the error through **GciErr**, and in the GemBuilder for Smalltalk interface you program the notificationBlock: in GSSession with actions to take upon notification. See your particular interface manual for details of its behavior in response to errors. To handle errors in the interface, RPC applications

need to call **GciPollForSignal** regularly to receive the error from the Gem. See your interface manual for more information.

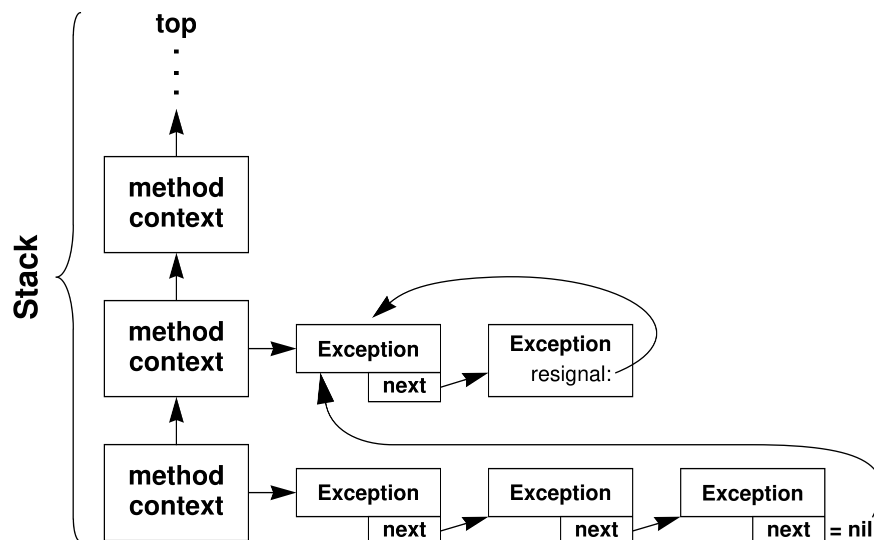
GemStone Smalltalk allows you to define two kinds of exceptions: *static exceptions* and *Activation exceptions*.

Activation Exceptions

A *activation exception* is associated with a method and the associated state in which the Smalltalk virtual machine is presently executing. These exceptions live and die with their associated method contexts—when the method returns, control is passed to the next method and the exception is gone.

Each exception is associated with one method context, but each method context can have a stack of associated exceptions. The relationship is diagrammed in Figure 11.1.

Figure 11.1 Method Contexts and Associated Exceptions



Static Exceptions

A *static exception* is a final line of defense—if you define one, it will take control in the event of any error for which no other handler has been defined. A static exception executes without changing in any way the stack, or the return value of the method that called it. Static exception handlers are therefore useful for

handling errors that appear at unpredictable times, like the errors listed in Table 11.1. You can use a static exception handler as you would an interrupt handler, coding it to change the value of some global variable, perhaps, so that you can determine that an error did, in fact, occur.

The errors in Table 11.1 are sometimes called *event errors*. Although they are not true errors, their implementation is based on the GemStone error mechanism. For examples that use these event errors, also called signals, see Chapter 11, "Signals and Notifiers".

Table 11.1 GemStone Event Errors

Name	Num ber	Description
#rtErrSignalAbort	6009	While running outside a transaction, Stone requested Gem to abort. This error is generated only if you have executed the method `enableSignaledAbortError`. No arguments.
#abortErrLostOtRoot	3031	While running outside a transaction, Stone requested Gem to abort. Gem did not respond in the allocated time, and Stone was forced to revoke access to the object table. No arguments.
#rtErrSignalCommit	6008	An element of the notify set was committed and added to the signaled objects set. This error is received only if you have executed the method `enableSignaledObjectsError`. No arguments.
#rtErrSingalGemStoneSession	6010	Your session received a signal from another GemStone session. This error is received only if you have executed the method `enableSignaledGemstoneSes-sionError`. Arguments: 1. The session ID of the session that sent the signal. 2. An integer representing the signal. 3. A message string.

Defining Exceptions

Instances of class Exception represent specific *exception handlers*—the code to execute in the event that the error occurs.

An exception handler—an instance of class Exception—consists of:

- an optional category to which the error belongs;
- an optional error number to further distinguish errors within a category;
- the code to execute in the event that the specific error is raised; and

in activation exception handlers, a pointer to the next exception handler associated with this method context, as shown in Figure 11.1 on page 11-6. If this pointer is nil, the interpreter searches the previous method context for its stack of exception handlers instead.

The interpreter decides to give control to a specific exception handler based upon its category and error number. These ideas are explained in detail below.

Categories and Error Numbers

Errors are defined in an instance of LanguageDictionary. Each LanguageDictionary represents a specific category of errors. Your application can include any number of such error dictionaries, each representing a given category of error. However, each such category of errors must be defined in UserGlobals or some other dictionary to which your application has access.

Like the SignalDict SymbolDictionary described earlier, the dictionary that defines your errors is keyed on symbols such as #English or #Kwakiutl that name natural languages. Each key is associated with an array of error-describing objects.

The index into the array is a specific error number, and the value is either a string or another array.

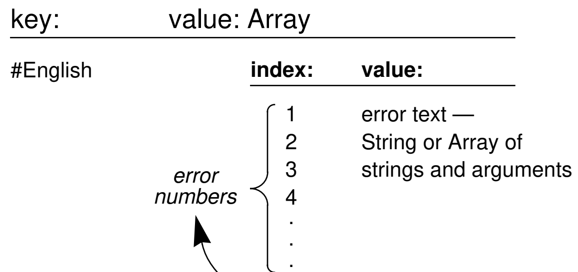
If it is a string, the string represents the text of an error message. Using an array, however, allows you to capture and report diagnostic information from the context of the error. This works just as it did using the signaling mechanism described earlier; SmallIntegers interspersed with strings act as indexes into an array of arguments passed back from the specific error that was raised. When the error string is constructed, each positive SmallInteger in the error object is replaced by the result of sending `asString` to the corresponding element of the `args:` array specified when the exception is raised. (This array is discussed in detail in the next section.)

The GemStone system itself uses this mechanism. GemStone errors are defined in the dictionary GemStoneError, and all GemStone system errors belong to this category. This dictionary is accessible to all users by virtue of being defined in the dictionary Globals. The dictionary GemStoneError contains one key: #English. The value of this key is an array.

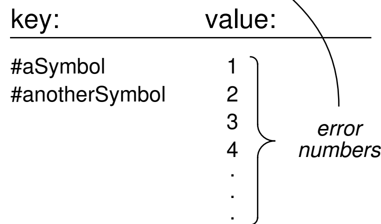
It is not, however, an array of error numbers. Numbers are not the most useful possible way to refer to errors; sprinkling code with numbers does not lead to an application that can be easily understood and maintained. For this reason, GemStoneError defines mnemonic symbols for each error number in a SymbolDictionary called ErrorSymbols. The keys in this dictionary are the mnemonic symbols, and their values are the error numbers. These numbers, in turn, are used to map each error to the appropriate index of the array that holds the error text. This structure is diagrammed in Figure 11.2.

Figure 11.2 Defining Error Dictionaries

LanguageDictionary (error category)



SymbolDictionary



If your application needs only one exception handler, you need not define your own error dictionary. You can instead use the generic error already defined for you in the GemStone ErrorSymbols dictionary as #genericError.

For example, suppose we define the class Cargo as follows:

Example 11.7

```
Object subclass: #Cargo
  instVarNames: #(#name #diet #kind)
  classVars: #()
  poolDictionaries: #()
  inDictionary: UserGlobals
  constraints: #[#[ #name, Symbol],
                  #[ #diet, Symbol], #[ #kind, Symbol]]
  instancesInvariant: false
  isModifiable: false
%
```

We give our new class the following instance creation method:

Example 11.8

```
classMethod: Cargo
named: aName diet: aDiet kind: aKind
   | result |

   result := self new.
   result name: aName.
   result diet: aDiet.
   result kind: aKind.
   ^result.
%
```

And we create accessing and updating methods for its instance variables:

Example 11.9

```
run
Cargo compileAccessingMethodsFor: (Cargo instVarNames).
^true
%
```

Now we can make some instances:

Example 11.10

```
run
UserGlobals at: #Sheep put:
     (Cargo named: #Sheep diet: #Vegetarian kind #animal).
UserGlobals at: #Cabbage put:
     (Cargo named: #Cabbage diet: #Photosynthesis kind: #plant).
UserGlobals at: #Wolf put:
     (Cargo named: #Wolf diet: #Carnivore kind: #animal).
^true
%
```

We wish all the errors in this application to belong to the category CargoErrors, so we define a LanguageDictionary by that name and a SymbolDictionary to contain the mnemonic symbols for its errors:

Example 11.11

```
run
UserGlobals at: #CargoErrors put: LanguageDictionary new.
UserGlobals at: #CargoErrorSymbols put: SymbolDictionary new.
^true
%
```

We then populate these dictionaries with error symbols and error text:

Example 11.12

```
run
| errorMsgArray |
CargoErrorSymbols at: #VegetarianError put: 1.
CargoErrorSymbols at: #CarnivoreError put: 2.
CargoErrorSymbols at: #PlantError put: 3.
errorMsgArray := Array new.
CargoErrors at: #English put: errorMsgArray.
errorMsgArray at: 1 put: #('Sheep can''t eat wolves!').
errorMsgArray at: 2 put: #('Wolves won''t eat cabbage!').
errorMsgArray at: 3 put: #('Cabbages don''t eat animals!').
^true
%
```

We can now define some more meaningful methods for Cargo:

Example 11.13

```
method: Cargo
eat: aCargo
^name , ' ate ' , (self swallow: aCargo) , '.'
%
method: Cargo
swallow: aFood
^aFood name
%
```

And finally, we can verify that our example so far works as we expect:

Example 11.14

```
Wolf eat: Sheep
'Wolf ate Sheep.'
```

Handling Exceptions

Keep the handler as simple as possible, because you cannot receive any additional errors while the handler executes. Normally your handler should never terminate the ongoing activity and change to some other activity.

To define an exception handler for an activation exception, use the class method `Exception category: number: do:`.

- The argument to the `category:` keyword is the specific error category of the error you wish to catch—the instance of LanguageDictionary in which the error is defined.
- The argument to the `number:` keyword is the specific error number you wish to catch.
- The argument to the `do:` keyword is a four-argument block you wish to execute when the error is raised.

  The first argument to the four-argument block is the instance of Exception you are currently defining.

  The second argument to the four-argument block is the error category, which can be nil.

  The third argument to the four-argument block is the error number, which can be nil.

  The fourth argument to the four-argument block is the information the error passes to the exception handler in the form of arguments.

If your exception handler does not specify an error number (an error number of nil), then it receives control in the event of any error of the specified category. If your exception handler does not specify a category (a category of nil), then it receives control in the event of any error at all. If your exception handler specifies an error number but the error category is nil, the error number is ignored and this exception handler receives control in the event of any error at all.

For example, the following exception handler defines #VegetarianError so that, when it is raised, it changes the result returned, changing the object eaten from Wolf to Cabbage:

Example 11.15

```
method: Cargo
eat: aCargo
   Exception
      category: CargoErrors
      number: (CargoErrorSymbols at: #VegetarianError)
      do: [:ex:cat:num:args | aCargo == Wolf
                  ifTrue: [ 'Cabbage' ] ].

^name , ' ate ' , (self swallow: aCargo) , '.'
%
```

To define an exception handler for static exceptions, use the Exception class method `installStaticException: category: number:` instead.

- The argument to the `installStaticException:` keyword is the block you wish to execute when the error is raised.
- The argument to the `category:` keyword is the specific error category of the error you wish to catch—the instance of LanguageDictionary in which the error is defined.
- The argument to the `number:` keyword is the specific error number you wish to catch.

The same rules about error categories and error numbers apply to static exceptions as to activation exceptions.

Raising Exceptions

To raise an exception, use the class method `System signal:args: signalDictionary:`.

- The argument to the `signal:` keyword is the specific error number you wish to signal.

The argument to the `args:` keyword is an array of information you wish to pass to the exception handler. This is the array whose elements can be used to build the error messages described in the section entitled “Defining Exceptions” on

page 11-9. The integers interspersed with strings in the error message defined with your exceptions index into this array. When the error string is constructed, each positive SmallInteger in the error object is replaced by the result of sending `asString` to the corresponding element of this array.

- The argument to the `signalDictionary:` keyword is the specific error category of the error you wish to signal—the instance of LanguageDictionary in which the error is defined.

To raise the generic exception defined for you in ErrorSymbols as #genericError, use the class method `System genericSignal: text: args:`, or one of its variants.

- The argument to the `genericSignal:` keyword is an object you can define to further distinguish between errors, if you wish. If you do not wish, it can be nil.
- The argument to the `text:` keyword is a string you can use for an error message. It will appear in GemStone's error message when this error is raised. It can be nil.

The argument to the `args:` keyword is an array of information you wish to pass to the exception handler. This is the array whose elements can be used to build the error messages described in the section entitled "Defining Exceptions" on page 11-9. The integers interspersed with strings in the error message defined with your exceptions index into this array. When the error string is constructed, each positive SmallInteger in the error object is replaced by the result of sending `asString` to the corresponding element of this array.

Other variants of this message are `System genericSignal: text: arg:` for errors having only one argument, or `System genericSignal: text:` for errors having no arguments.

For example, we can now raise the exception #VegetarianError for which we defined a handler in the previous example:

Example 11.16

```
method: Cargo
swallow: aFood
diet = #Vegetarian ifTrue: [
     aFood kind = #plant ifFalse: [
           ^System signal: (CargoErrorSymbols at: #VegetarianError)
                 args: #() signalDictionary: CargoErrors
     ]
].
^aFood name
%
```

When we test this exception, we get:

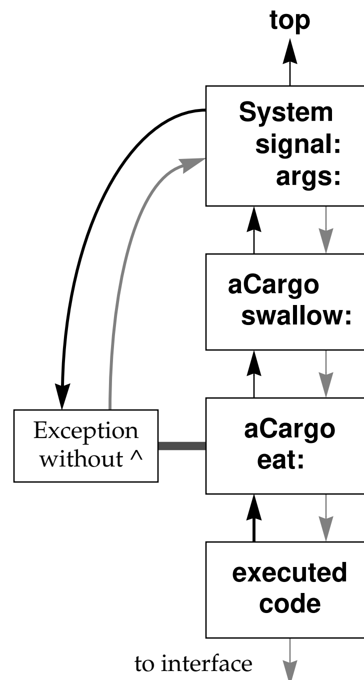
Example 11.17

```
run
Sheep eat: Wolf
'Sheep ate Cabbage.'
%
```

Flow of Control

Exception handlers with no explicit return operate like interrupt handlers—they return control directly to the method from which the exception was raised. All static exception handlers should be written this way, because the stack usually changes by the time they catch an error. Activation exception handlers can also be written to behave that way, like the one in Example 11.15. So in Examples 11.16 and 11.17 control returns directly to the method `swallow:` as shown in Figure 11.3.

Figure 11.3 Default Flow of Control in Exception Handlers



Sometimes, however, this is not useful behavior—the application may simply have to raise the same error again. In activation exception handlers it can be useful instead to return control to the method that defined the handler, such as method `eat:` in Example 11.15.

You can accomplish this by defining an explicit return (using the return character ^) in the block that is executed when the exception is raised. For example, the following method redefines how the exception #VegetarianError is to be handled. It explicitly returns a string. Code that follows after this exception is raised is therefore never executed, because control returns to the sender of this message instead:

Example 11.18

```
method: Cargo
eat: aCargo
    Exception
         category: CargoErrors
         number: (CargoErrorSymbols at: #VegetarianError)
         do: [:ex:cat:num:args | ^ 'The sheep is not hungry.' ].
    ^name + ' ate ' + (self swallow: aCargo)
%
```

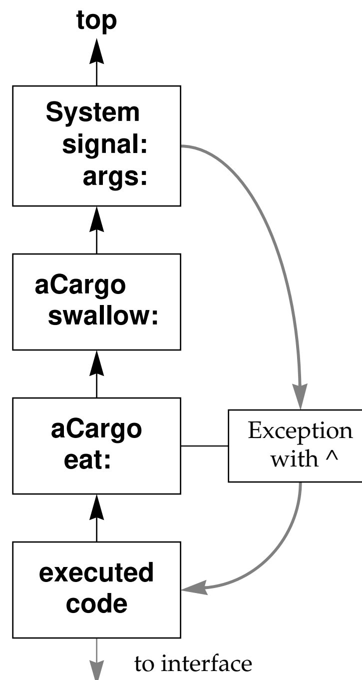
If we leave the method `swallow:` as defined as in Example 11.16, then the code below produces the following result:

Example 11.19

```
run
Sheep eat: Wolf
'The sheep is not hungry.'
%
```

Figure 17.4 shows the flow of control in Examples 11.18 and 11.19.

Figure 11.4 Activation Exception Handler With Explicit Return



When you raise an error in a user action, you need to install an exception handler that explicitly returns, or the exception block may not leave the activation record stack in the correct state for continued execution. If the exception block does not contain an explicit return, the call to userAction should be placed by itself inside a method similar to this:

Example 11.20

```
callAction: aSymbol withArgs: args
^ System userAction: aSymbol withArgs: args
%
```

Signaling Other Exception Handlers

Under certain circumstances, your exception handler can choose to pass control to a previously defined exception handler, one that is below the present exception handler on the stack. To do so, your exception handler can send the message `resignal: number: args:`.

- The argument to the `resignal:` keyword is the specific error category of the error you wish to signal—the instance of LanguageDictionary in which the error is defined.
- The argument to the `number:` keyword is the specific error number you wish to signal.
- The argument to the `args:` keyword is an array of information you wish to pass to the exception handler. This is the array whose elements can be used to build the error messages described above. The integers interspersed with strings in the error message defined with your exceptions index into this array. When the error string is constructed, each positive SmallInteger in the error object is replaced by the result of sending `asString` to the corresponding element of this array.

For example, imagine we compile a method that defines an exception handler as follows:

Example 11.21

```
method: Cargo
eat: aCargo
     Exception
            category: CargoErrors
            number: nil
            do: [:ex:cat:num:args |
                   (num == (CargoErrorSymbols at: #VegetarianError))
                         ifTrue: [ex resignal: cat number: num args:
args]
                         ifFalse: [ ^ 'The sheep is not hungry.' ]
                         ].
     ^name + ' ate ' + (self swallow: aCargo)
%
```

We then execute the following code:

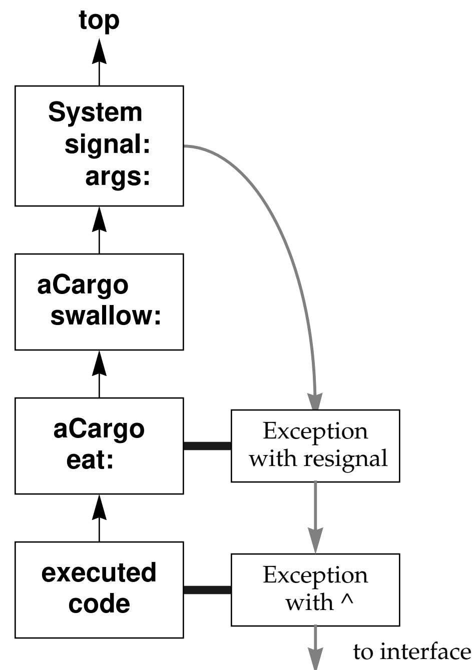
Example 11.22

```
run
    Exception
        category: CargoErrors
        number: nil
        do: [:ex:cat:num:args |
            ^'Shepherd intervened with a resignal.' ].
    Sheep eat: Wolf.
%
'Shepherd intervened with a resignal.'.
```

The `resignal:` message in the previously defined method (see Example 17.21) means that, when the VegetarianError is raised, control passes to the exception handler defined in the executed code instead. This means that the result of executing `Sheep eat: Wolf` will be a return of the string `'Shepherd intervened with a resignal.'`.

Figure 11.5 shows the flow of control in Examples 11.21 and 11.22.

Figure 11.5 Activation Exception Handler Passing Control to Another Handler



Removing Exception Handlers

You can define an exception so that it removes itself after it has been raised, using the Exception instance method `remove`. In conjunction with the `resignal:` mechanism described above, `remove` allows you to set up your application so that successive occurrences of the same error (or category of errors) are handled by successively older exception handlers that are associated with the same context.

For example, suppose we define the `eat:` method as shown in Example 17.21 and then execute the following code:

Example 11.23

```
run
Exception
   category: CargoErrors
   number: nil
   do: [:ex:cat:num:args |ex remove.'clover' ].
Exception
   category: CargoErrors
   number: nil
   do: [:ex:cat:num:args | ex remove.'grass' ].
^(Sheep eat: Wolf) + ', ' + (Sheep eat: Wolf) + '.'
%
'Sheep ate grass, Sheep ate clover.'
```

The first occurrence of VegetarianError executes the most recent exception defined, which returns the string `'grass'`. The exception then removes itself, so that the next occurrence of the same error executes the exception handler stacked previously within the same method context. This exception handler returns the string `'clover'`.

Recursive Errors

If you define an exception handler broadly to handle many different errors, and you make a programming mistake in your exception handler, the exception handler may then raise an error that calls itself repeatedly. Such infinitely recursive error handling eventually reaches the stack limit. The resulting stack overflow error is received by whichever interface you are using.

If you receive such an error, check your exception handler carefully to determine whether it includes errors that are causing the problem.

Uncontinuable Errors

Some errors are sufficiently complex or serious that execution cannot continue. Exception handlers cannot be defined for these errors—instead, control returns to whichever interface you are using.

Table 11.2 lists uncontinuable errors.

Table 11.2 Uncontinuable Errors

bkupErrRestoreSuccessful
abortErrFinishedObjAudit
rtErrStep
rtErrCommitAbortPending
rtErrHardBreak
rtErrStackLimit
rtErrUncontinuable
rtErrMethodBreakpoint
rtErrMessageBreakpoint

Chapter 12 Tuning Performance

Smalltalk includes several tools to help you tune your applications for faster performance.

Clustering Objects for Fast Retrieval
discusses how you can cluster objects that are often accessed together so that many of them can be found in the same disk access. Unnecessarily frequent disk access is ordinarily the worst culprit when application performance is not up to expectations.

Optimizing for Faster Execution
describes the profiling tool that allows you to pinpoint the problem areas in your application code and rewrite it to use more efficient mechanisms.

Modifying Cache Sizes for Better Performance
explains how to increase or decrease the size of various caches in order to minimize disk access and storage reclamation, both of which can significantly slow your application.

Generating Native Code
defines how and when native code is generated for your application.

12.1 Clustering Objects for Faster Retrieval

As you've seen, GemStone ordinarily manages the placement of objects on the disk automatically—you're never forced to worry about it. Occasionally, you might choose to group related objects on secondary storage to enable GemStone to read all of the objects in the group with as few disk accesses as possible.

Because an access to the first element usually presages the need to read the other elements, it makes sense to arrange those elements on the disk in the smallest number of disk pages. This placement of objects on physically contiguous regions of the disk is the function of class Object's *clustering* protocol. By clustering small groups of objects that are often accessed together, you can sometimes improve performance.

Clustering a group of objects packs them into disk pages, each page holding as many of the objects as possible. The objects are contiguous within a page, but pages are not necessarily contiguous on the disk.

Although different kinds of objects require different amounts of disk space, it's not unreasonable to take as a rule of thumb that a page can accommodate about 100–250 simple pointer objects with up to eight instance variables each.

Will Clustering Solve the Problem?

Clustering objects solves a specific problem—slow performance due to excessive disk accessing. However, disk access is not the only factor in poor performance. In order to determine if clustering will solve your problem, you need to do some diagnosis. GemStone provides methods in class System to tell you how many times your application is accessing the disk. To find out how many pages your session has read from the disk since the session began, execute:

```
System pageReads
```

To find out how many pages your session has written to the disk since the session began, execute:

```
System pageWrites
```

You can use these methods to build a *tracer* into your application. You execute these expressions before you commit each transaction to learn whether your application has written large temporary objects to disk, or to discover how many pages it read in order to perform a particular query. You then execute these expressions after you commit each transaction to determine the number of disk accesses required by the process of committing the transaction.

For example, the Topaz script in Example 12.1 reveals how many disk pages were written as a result of committing a transaction (represented by *your Smalltalk code*). Because these methods tell you how many pages were written to disk since the beginning of the session, you must subtract the first result from the second in order to determine the disk accesses performed only by the most recent `commitTransaction`.

Example 12.1

```
run
your Smalltalk DB code
%
run
System pageWrites
%
run
System commitTransaction
%
run
System pageWrites
%
```

It is tempting to ignore these issues until you experience a problem such as an extremely slow application, but of you keep track of such statistics on a regular (even if intermittent) basis, you will have a better idea of what is “normal” behavior when a problem crops up.

Cluster Buckets

You can think of clustering as writing the components of their receivers on a stream of disk pages. When a page is filled, another is randomly chosen and subsequent objects are written on the new page. A new page is ordinarily selected for use only when the previous page is filled, or when a transaction ends. Sending the message `cluster` to objects in repeated transactions will, within the limits imposed by page capacity, place its receivers in adjacent disk locations. (Sending the message cluster to objects repeatedly within a transaction has no effect.)

The stream of disk pages used by `cluster` and its companion methods is called a *bucket*. GemStone captures this concept in the class ClusterBucket.

If you determine that clustering will improve your application’s performance, you can use instances of the class ClusterBucket to help. All objects assigned to the

same instance of ClusterBucket are to be clustered together. When the objects are written, they are moved are moved to contiguous locations on the same page, if possible. Otherwise the objects are written to contiguous locations on several pages.

Once an object has been clustered into a particular bucket and committed, that bucket remains associated with the object until you specify otherwise. When the object is modified, it continues to cluster with the other objects in the same bucket, although it might move to another page within the same bucket.

Cluster Buckets and Extents

Cluster buckets can be defined to cluster objects on a specific extent—either a file in the file system or a raw partition, a range of physical blocks on the disk. For this purpose, they define an instance variable called *extentId*. If you do not specify an extent, pages are allocated to the disk as they ordinarily would be. In other words, an *extentId* of nil specifies that you don't care to which extent objects are clustered.

NOTE:
Cluster buckets for which no extent is specified are allocated pages according to the DBF_ALLOCATION_MODE parameter of the Stone configuration file. For details on this parameter, see the GemStone System Administration Guide.

If you wish to know what extents are available, execute the expression `SystemRepository fileSizeReport`. This expression returns a string describing the extents that are available to you, as well as other information. For example:

```
SystemRepository fileSizeReport
```

returns a report of the form:

```
'Extent #1
-----------
   Filename = !TCP@servio#dbf!/servio1/user/davidm/extent0.dbf
   Replicate = NONE
   File size =       20.0 Megabytes
   Space available = 0.23 Megabytes
Totals
------
   Repository size = 20.0 Megabytes
   Free Space =      0.23 Megabytes
'
```

Extents can be of varying sizes. If your system administrator has specified no upper limit on the size of a given extent, then it can grow until the disk is filled. If the extent is already full and cannot accommodate the objects you wish to cluster, then the specified extent is ignored and objects are clustered on another extent.

You can determine the extentId of a given cluster bucket by executing an expression of the form:

```
aClusterBucket extentId
```

You can change the extentId of a cluster bucket by executing an expression of the form:

```
aClusterBucket extentId: 2
```

Assuming that aClusterBucket previously used extent 1, the expression above changes the extent to which objects clustered in aClusterBucket will be written, when they are next written to disk. If you then commit this transaction, objects clustered in aClusterBucket will move to extent 2 when they are next modified by subsequent transactions.

Using Existing Cluster Buckets

By default, a global array called AllClusterBuckets defines seven instances of ClusterBucket. Each can be accessed by specifying its offset in the array. For example, the first instance, `AllClusterBuckets at: 1`, is the default bucket when you log in. It specifies an *extentId* of nil. This bucket is invariant—you cannot modify it.

The second, third, and seventh cluster buckets in the array also specify an *extentId* of nil. They can be used for whatever purposes you require and can all be modified.

The GemStone system makes use of the fourth, fifth, and sixth buckets of the array AllClusterBuckets:

- The fourth bucket in the array is the bucket used to cluster the methods associated with kernel classes.
- The fifth bucket in the array is the bucket used to cluster the strings that define source code for kernel classes.
- The sixth bucket in the array is the bucket used to cluster other kernel objects such as globals.

You can determine how many cluster buckets are currently defined by executing:

```
System maxClusterBucket
```

A given cluster buckets offset in the array specifies its *clusterId*. A cluster bucket's clusterId is an integer in the range of 1 to `(System maxClusterBucket)`.

NOTE:
For compatibility with previous versions of GemStone, you can use a clusterId as an argument to any keyword that takes an instance of ClusterBucket as an argument.

You can determine which cluster bucket is currently the system default by executing:

```
System currentClusterBucket
```

You can access all instances of cluster buckets in your system by executing:

```
ClusterBucket allInstances
```

You can change the current default cluster bucket by executing an expression of the form:

```
System clusterBucket: aClusterBucket
```

Creating New Cluster Buckets

You are not limited to these predefined instances. With write authorization to the DataCurator segment, you can create new instances of ClusterBucket with the simple expression:

```
ClusterBucket new
```

This expression creates a new instance of ClusterBucket and adds it to the array AllClusterBuckets (which resides on the DataCurator segment). You can then access the bucket in one of two ways. You can assign it a name:

```
UserGlobals at: #empClusterBucket put: (ClusterBucket new)
```

You could then refer to it in your application as empClusterBucket. Or you can use the offset into the array AllClusterBuckets. For example, if this is the first cluster bucket you have created, you could refer to it this way:

```
AllClusterBuckets at: 8
```

You can determine the clusterId of a cluster bucket by sending it the message `clusterId`. For example:

```
empClusterBucket clusterId
8
```

You can access an instance of ClusterBucket with a specific clusterId by sending it the message `bucketWithId:`. For example:

```
ClusterBucket bucketWithId: 8
empClusterBucket
```

You can create a new cluster bucket and specify the extent it must use for clustering. For example, the following expression creates a new instance of cluster bucket whose objects will be clustered in extent 3, and adds it to the array AllClusterBuckets.

```
ClusterBucket newForExtent: 3
```

Arguments to the `newForExtent:` keyword must be in the range of 1 to `(SystemRepository numberOfExtents)`.

You can create and use as many cluster buckets as you need, up to thousands, if necessary.

NOTE:
For best performance and disk space usage, use no more than 32 cluster buckets in a single session.

Cluster Buckets and Concurrency

Cluster buckets are designed to minimize concurrency conflicts. As many users as necessary can cluster objects at the same time, using the same cluster bucket, without experiencing concurrency conflicts. Each cluster operation only reads the associated cluster bucket.

However, creating a new instance of ClusterBucket automatically adds it to the global array AllClusterBuckets. Adding an instance to AllClusterBuckets causes a concurrency conflict when more than one transaction tries to create new cluster buckets at the same time, since all the transactions are all trying to write the same array object.

You should design your clustering when you design your application to avoid concurrency conflicts. Create all the instances of ClusterBucket you anticipate needing and commit them in one or few transactions.

To facilitate this kind of design, GemStone allows you to associate descriptions with specific instances of ClusterBucket. In this way, you can communicate to your fellow users the intended use of a given cluster bucket with the message `description:`. For example:

Example 12.2

```
UserGlobals at: #empClusterBucket put: (ClusterBucket new)
empClusterBucket description: 'Use this bucket for
      clustering employees and their instance variables.'
```

As you can see, the message `description:` takes a string of text as an argument.

Changing the attributes of a cluster bucket, such as its description, clusterId, or extentId, writes that cluster bucket and can cause concurrency conflict. Only change these attributes when necessary.

NOTE:
For best performance and disk space usage as well as avoiding concurrency conflicts, create the required instances of ClusterBucket all at once, instead of on a per-transaction basis, and update their attributes infrequently.

Cluster Buckets and Indexing

Indexes on a nonsequenceable collection are created and modified using the cluster bucket associated with the specific collection, if any. To change the clustering of a nonsequenceable collection:

1. Remove its index.
2. Recluster the collection.
3. Re-create its index.

Clustering Objects

Class Object defines several clustering methods. One method is simple and fundamental, and another is more sophisticated and attempts to order the receiver's instance variables as well as writing the receiver itself.

The Basic Clustering Message

The basic clustering message defined by class Object is `cluster`. For example:

```
myObject cluster
```

This simplest clustering method simply assigns the receiver to the current default cluster bucket—it does not attempt to cluster the receiver's instance variables.

When the object is next written to disk, it will be clustered according to the attributes of the current default cluster bucket.

If you wish to cluster the instance variables of an object, you can define a special method to do so.

CAUTION:
Do not redefine the method `cluster` in the class Object, because other methods rely on the default behavior of the `cluster` method. You can, however, define a `cluster` method for classes in your application if required.

Suppose, for example, that you defined class Name and class Employee like this:

Example 12.3

```
Object subclass: 'Name'
       instVarNames: #('first' 'middle' 'last')
       classVars: #( )
       poolDictionaries: #[ ]
       inDictionary: UserGlobals
       constraints: #( )
       isInvariant: false.
Object subclass: 'Employee'
       instVarNames: #('name' 'job' 'age' 'address')
       classVars: #( )
       poolDictionaries: #[]
       inDictionary: UserGlobals
       constraints: #[
                      #['name', Name],
                      #['job', String],
                      #['age', SmallInteger],
                      #['address', String]
                     ]
       isInvariant: false.
```

The method shown in Example 12.4 might be a suitable clustering method for class Employee. (A more purely object-oriented approach would embed the information on clustering first, middle, and last names in the cluster method for name, but such an approach does not exemplify the breadth-first clustering technique we wish to show here.)

Example 12.4

```
method: Employee
clusterBreadthFirst
  self cluster.
  name cluster.
  job cluster.
  address cluster.
     name first cluster.
     name middle cluster.
     name last cluster.
  ^false
```

The elements of byte objects such as instances of String and Float are always clustered automatically. A string's characters, for example, are always written contiguously within disk pages. Consequently, you need not send `cluster` to each element of each string stored in *job* or *address*—clustering the strings themselves is sufficient. Sending `cluster` to individual atomic objects such as characters, Booleans, SmallIntegers, and UndefinedObjects has no effect. Hence no clustering message is sent to *age* in the previous example.

The result of executing `Employee cluster` might look like this:

```
anEmp aName job address first middle last
```

`cluster` returns a Boolean value. You can use that value to eliminate the possibility of infinite recursion when you're clustering the variables of an object that can contain itself. Here are the rules that `cluster` follows in deciding what to return:

- If the receiver has already been clustered during the current transaction or if the receiver is an atomic object, `cluster` declines to cluster the object and returns true to indicate that all of the necessary work has been done.
- If the receiver is a byte object that has not been clustered in the current transaction, `cluster` writes it on a disk page and, as in the previous case, returns true to indicate that the clustering process is finished for that object.
- If the receiver is a pointer object that has not been clustered in the current transaction, `cluster` writes the object and returns false to indicate that the receiver might have instance variables that could benefit from clustering.

Clustering and Authorization

You must have write authorization for an object's segment in order to cluster the object.

Depth-first Clustering

`clusterDepthFirst` differs from `cluster` only in one way: it traverses the tree representing its receiver's instance variables (named, indexed, or unordered) in depth-first order, assigning each node to the current default cluster bucket as it's visited. That is, it writes the receiver's first instance variable, then the first instance variable of that instance variable, then the first instance variable of that instance variable, and so on to the bottom of the tree. It then backs up and visits the nodes it missed before, repeating the process until the whole tree has been written.

This method clusters an Employee as shown below:

```
anEmp aName first middle last job address
```

Assigning Cluster Buckets

The clustering methods described above use the current default cluster bucket. If you wish to use a specific cluster bucket instead, you can use the method `clusterInBucket:`. For example, the following expressions clusters aBagOfEmployees using the specific cluster bucket empClusterBucket:

```
aBagOfEmployees clusterInBucket: empClusterBucket
```

In order to determine the cluster bucket associated with a given object, you can send it the message `clusterBucket`. For example, after executing the example above, the following example would return the value shown below:

```
aBagOfEmployees clusterBucket
empClusterBucket
```

Clustering vs. Writing to Disk

The clustering methods described so far specify that objects are to be clustered when they are next written to disk. Ordinarily, this happens when you commit the transaction, and not before. If you wish to write certain objects to disk immediately, send them the message `moveToDisk` or `moveToDiskInBucket:`. The latter message takes as its argument an instance of ClusterBucket.

Using Several Cluster Buckets

The availability of several cluster buckets instead of a single one is convenient when you want to write a loop that clusters parts of each object in a group into

separate pages. Suppose that you had defined class SetOfEmployees and class Employee as in Chapter 4, “Collection and Stream Classes.” Suppose, in addition, that you wanted a clustering method to write all employees contiguously and then write all employee addresses contiguously. If you had only one cluster bucket at your disposal, you would need to define your clustering method this way:

Example 12.5

```
method: SetOfEmployees
clusterEmployees
  self do: [:n | n cluster].
  self do: [:n | n address cluster].
```

With multiple buckets, you can write the method as a single loop, as shown in the following example (which assumes that the currentClusterBucket and the maxClusterBucket are not the same):

Example 12.6

```
method: SetOfEmployees
clusterEmployees
self do: [:n | n clusterInBucket: System currentClusterBucket.
               n address clusterInBucket: System maxClusterBucket.
         ].
```

In the previous version of this method, each employee had to be fetched once for clustering, then fetched again in order to cluster the employee’s address. The new version is probably more efficient despite its greater complexity, because it fetches each employee only once.

Clustering Class Objects

Clustering provides the most benefit for small groups of objects that are often accessed together. It happens that a class with its instance variables is just such a group of objects. Those instance variables of a class that describe the class’s variables and their constraints are often accessed in a single operation, as are the instance variables that contain a class’s methods. Therefore, class Behavior defines the following special clustering methods for classes:

Table 12.1 Clustering Protocol

`clusterBehavior`	Clusters in depth-first order the parts of the receiver required for executing Smalltalk code (the receiver and its method dictionary). Returns true if the receiver was already clustered in the current transaction.
`clusterDescription`	Clusters in depth-first order those instance variables in the receiver that describe the structure of the receiver's instances. (Does not cluster the receiver itself.) The instance variables clustered are *instVarNames*, *classVars*, *categories*, *class histories*, and *constraints*.
`clusterBehaviorExceptMethods:` *aCollectionOfMethodNames*	This method can sometimes provide a better clustering of the receiving class and its method dictionary by omitting those methods that are seldom used. This omission allows often-used methods to be packed more densely.

The following code clusters class Employee's structure-describing variables, then its class methods, and finally its instance methods.

Example 12.7

```
| behaviorBucket descriptionBucket |
behaviorBucket := AllClusterBuckets at: 4.
descriptionBucket := AllClusterBuckets at: 5.
System clusterBucket: descriptionBucket.
Employee clusterDescription.
System clusterBucket: behaviorBucket.
Employee class clusterBehavior.
Employee clusterBehavior.
```

The next example clusters all of class Employee's instance methods except for `address` and `address:`

```
Employee clusterBehaviorExceptMethods: #(#address #address:).
```

Maintaining Clusters

Once you have clustered certain objects, they do not necessarily stay that way forever. You may therefore wish to check an object's location, especially if you suspect that such declustering is causing your application to run more slowly than it used to.

Determining an Object's Location

To enable you to check your clustering methods for correctness, Class Object defines the message `page`, which returns an integer identifying the disk page on which the receiver resides. For example:

```
anEmp page
2539
```

Disk page identifiers are returned only for temporary use in examining the results of your custom clustering methods—they are not stable pointers to storage locations. The page on which an object is stored can change for several reasons, discussed in the next section.

For atomic objects (instances of SmallInteger, Character, Boolean, or UndefinedObject) the page number returned is 0.

To enable you to check whether your object is in the correct extent, Class Repository defines the message `extentForPage`, which returns an integer identifying the extent in which a given disk page resides. The following example determines the extent for page 2539, as revealed above:

```
SystemRepository extentForPage: (anEmp page)
7
```

If you need to know how many extents are available to you, execute the expression:

```
SystemRepository numberOfExtents
```

Why Do Objects Move?

The page on which an object is stored can change for any of the following reasons:

- A clustering message is sent to the object or to another object on the same page.
- The current transaction is aborted.
- The object is modified.
- Another object on the page with the object is modified.
- The extent in which you requested the object be clustered had insufficient room.

As your application updates clustered objects, new values are placed on secondary storage using GemStone's normal space allocation algorithms. When objects are moved, they are automatically reclustered within the same clusterId. If a specific clusterId was specified, it continues to be used; if not, the default clusterId is used.

If, for example, you replace the string at position 2 of the clustered array ProscribedWords, the replacement string is stored in a page separate from the one containing the original, although it will still be within the same clusterId. Therefore, it might be worthwhile to recluster often-modified collections occasionally to counter the effects of this fragmentation. You'll probably need some experience with your application to determine how often the time required for reclustering is justified by the resulting performance enhancement.

12.2 Optimizing for Faster Execution

As stated earlier in the chapter, disk access ordinarily has the greatest impact on application performance. However, your Smalltalk code can also affect the speed of your application—as with other programming languages, some code is more efficient than other code. In order to help you determine how you can best optimize your application, Smalltalk provides a profiling tool, defined by the class ProfMonitor.

The Class ProfMonitor

The class ProfMonitor allows you to sample the methods that are executed in a given block of code and to estimate the percentage of total execution time represented by each method. ProfMonitor starts a timer that determines which method is executing at specified intervals for a specified period of time. When it is done, it collects the results and returns them in the form of a string formatted as a report.

To do so, ProfMonitor stores the results temporarily in a file. The default file name used for this file is /tmp/gemprofile.tmp. If you wish, you can use ProfMonitor's instance method `fileName:` to specify that a different file name be used instead.

You can also specify the interval at which ProfMonitor checks to see which method is executing. By default, ProfMonitor checks execution every 10 ms. You can use ProfMonitor's instance method `interval:` to specify a shorter or longer interval in milliseconds.

By default, ProfMonitor reports every method it found executing, even once. If you are interested only in methods that execute a certain number of times or more, you can change the lower bound on this tally. ProfMonitor's instance method `reportDownTo:` *anInteger* allows you to specify that methods executed fewer times than the argument are to be omitted from the report.

Profiling Your Code

To determine the execution profile for a piece of code, you must format it as a block. It can then be provided as the argument to `monitorBlock:`. Example 12.8 uses the default interval of 10 ms and includes every method it finds in its results, even those executing only once:

Example 12.8

```
ProfMonitor monitorBlock: [ 10 timesRepeat:
      [ System myUserProfile dictionaryNames ] ]
```

As a convenience to Smalltalk programmers, the method `spyOn:` is also available to perform the same function as `monitorBlock:`. Example 12.9 is exactly equivalent to 12.8.

Example 12.9

```
ProfMonitor spyOn: [ 10 timesRepeat:
      [ System myUserProfile dictionaryNames ] ]
```

Example 12.10 uses a variant of `monitorBlock:` to check every 20 ms and include only methods that were found executing at least five times.

Example 12.10

```
ProfMonitor
      monitorBlock: [ 10 timesRepeat:
            [ System myUserProfile dictionaryNames ]]
      downTo: 5
      interval: 20
```

Profiling can be started and stopped using the instance methods `startMonitoring` and `stopMonitoring`. The instance method `gatherResults` tallies the methods that were found, and the instance method `report` returns a string formatting the results. Using these methods, you can profile any arbitrary sequence of Smalltalk statements; they need not be a block. Example 12.11 creates a new instance of ProfMonitor and changes the default file it will use to tally the results. It then starts profiling, executes the code to be profiled, stops profiling, tallies the methods encountered, and reports the results.

Example 12.11

```
| aMonitor |
aMonitor := ProfMonitor newWithFile: 'profile.tmp'.
aMonitor startMonitoring.
10 timesRepeat: [ System myUserProfile dictionaryNames ].
aMonitor stopMonitoring; gatherResults; report.
```

The class method `profileOn` also starts profiling. You can use it, for example, to profile code contained in a file-in script. Example 12.12 shows how to do this using Topaz format.

Example 12.12

```
! get a profile report on a filein script
run
UserGlobals at: #Monitor put: ProfMonitor profileOn
%
input testFileScript.gs
! turn off profiling and get a report
run
Monitor profileOff
%
```

If you simply want to know how long it takes a given block to return its value, you can use the familiar Smalltalk method `millisecondsToRun:`, defined in class System. It takes a zero-argument block as its argument and returns the time in milliseconds required to evaluate the block.

The Profile Report

The profiling methods discussed in Examples 12.8 through 12.12 return a string formatted as a report. The report has three sections, as shown in the sample run:

Figure 12.1

```
 topaz 1> run
ProfMonitor monitorBlock:[
  10 timesRepeat:[ System myUserProfile dictionaryNames]
  ]
%
STATISTICAL SAMPLING RESULTS
elapsed cpu time:    420 ms
monitoring interval: 10 ms

 tally       %   class and method name
------   -----   --------------------------------------
    35   83.33   IdentityDictionary        >> associationsDo:
     4    9.52   AbstractDictionary        >>
associationsDetect:ifNone:
     2    4.76   SymbolList                >> namesReport
     1    2.38   SymbolList                >> names
    42  100.00   Total

STATISTICAL METHOD SENDERS RESULTS
elapsed cpu time:    420 ms
monitoring interval: 10 ms

     %   tally  class and method name
------  ------  --------------------------------------
 83.3%      35  IdentityDictionary >> associationsDo:
            35 times sender was          AbstractDictionary >>
associationsDetect:ifNone:
           ------
            35 times receiver class was  SymbolDictionary
-----------------------------

  9.5%       4  AbstractDictionary >> associationsDetect:ifNone:
             3 times sender was          IdentityDictionary >>
associationsDo:
           ------
```

```
             3 times receiver class was  ComplexBlock
-----------------------------

  4.8%       2  SymbolList   >> namesReport
             2 times sender was          UserProfile  >>
dictionaryNames
           ------
             2 times receiver class was  SymbolList
-----------------------------

  2.4%       1  SymbolList   >> names
           ------
-----------------------------
OBJECT CREATION PROFILING Not Enabled

METHOD INVOCATION COUNTS

 tally  class and method name
------  -------------------------------------
 10660  Object                    >> _at:
  4367  ComplexBlock              >> value:
  2660  String                    >> at:put:
  1960  Association               >> value
  1872  Object                    >> at:
  1113  String                    >> addAll:
   976  SmallInteger              >> >
   793  Object                    >> size
   654  String                    >> size
   542  SmallInteger              >> <
   510  ProfMonEntry              >> tally
   453  Behavior                  >> new
   353  Behavior                  >> new:
   345  String                    >> add:
   330  Object                    >> class
   327  Array                     >> add:
   320  ProfMonEntry              >> tally:
   306  CharacterCollection       >> width:
   306  Number                    >> abs
   297  SequenceableCollection    >> atAllPut:
   297  CharacterCollection       >> speciesForConversion
   275  Object                    >> _basicSize
   264  SmallInteger              >> \\
```

```
 231  Object                      >> at:put:
 213  IdentityKeyValueDictionary >> hashFunction:
 213  Object                      >> identityHash
 213  KeyValueDictionary          >> keyAt:
 209  Object                      >> _basicAt:
 209  KeyValueDictionary          >> valueAt:
.
.
.
```

As you can see, the report lists the methods that the profile monitor encountered when it checked the execution stack every 10 ms. It sorts the methods according to the number of times they were found, with the most-often-used methods first. The ProfMonitor also calculates the percentage of total execution time represented by each method—useful information if you need to know where optimizing can do you the most good.

Optimization Hints

While optimization is an application-specific problem, we can provide a few ideas for improving application performance:

- Arrays tend to be faster than sets. If you do not need the particular semantics that a set affords, use an array instead.
- Prefer integers to floating-point numbers.
- Avoid coercing integers to floating point numbers. Although Smalltalk can easily handle mixing integers and floating point numbers in computations, the coercion required can be time-consuming.
- If you create an instance of a Dictionary class (or subclass) that you intend to load with values later, create it to be approximately the final required size in order to avoid rehashing, which can significantly slow performance.
- Prefer methods that invoke primitives, if possible, or methods that cause primitives to be invoked after fewer intermediate message-sends. (See the *GemStone C Interface* manual for information on writing your own primitive methods.)
- Prefer message-sends over path notation, where possible. (This is not possible for associative access, however.)

- Use the linkable interface when possible. Interfaces that run remotely incur interprocess communication overhead.
- Avoid assigning membership in more than 2000 groups to a given user.
- Avoid having a given segment participate in authorizations for more than 2000 groups.
- Prefer simpler blocks to more complex blocks. The most efficient blocks refer only to one or more literals, global variables, pool variables, class variables, local block arguments, or block temporaries; they also do not include a return statement.

  Less efficient blocks include a return statement and can also refer to one or more of the pseudovariables *super* or *self*, instance variables of *self*, arguments to the enclosing method, temporary variables of the enclosing method, block arguments, or block temporaries of an enclosing block.

  The least efficient blocks enclose a less efficient block of the kind described in the above paragraph.

  NOTE
  Blocks provided as arguments to the methods `ifTrue:`, `ifFalse:`, `ifTrue:ifFalse:`, `ifFalse:ifTrue:`, `whileFalse:`, *and* `whileTrue:` *are specially optimized. Unless they contain block temporary variables, you need not count them when counting levels of block nesting.*

- Avoid concatenating strings, instead use the `add:` method to dynamically resize strings. This is much more efficient.
- If you have a choice between a method that modifies an object and one that returns a modified copy, use the method that modifies the object directly if your application allows it. This creates fewer temporary objects whose storage will have to be reclaimed. For example, `String ,` creates a new string to use in modifying the old one, whereas `String add:` modifies a string.
- Avoid generating temporary objects whose storage will need to be reclaimed. Storage reclamation can slow your application significantly. The class Repository defines an instance method `findDisconnectedObjects`, whose purpose is to help determine the kinds of objects that are dead in the repository and thus allow application developers to learn ways to prevent unnecessary objects from being stored on disk. This method returns an array containing a list of objects that are not directly or indirectly connected to any permanent object. (This array is based on the state of the repository as viewed

by the transaction in which it is executed.) These objects are most likely no longer used and could be reclaimed by executing `markForCollection`.

CAUTION
To avoid having dead objects be inadvertently connected to the repository, disconnect this array after examining it.

NOTE
In order to execute the method `findDisconnectedObjects` *you must have GarbageCollection privileges.*

- Keep repository files on a disk reserved for their use, if possible. Particularly avoid putting repository files on the disk used for swapping.
- Commit transactions often enough so that the amount of new or modified data created in a transaction can fit within the shared page cache. For a rough estimate of the amount of space in the shared page cache available for your transactions, divide the size of the shared page cache by the number of users currently on the system who are also committing large transactions.

  The commit operation incurs a certain amount of overhead—it takes roughly between 50 and 1000 times longer than an operation on an individual data item. Therefore, if you are manipulating large amounts of data, such as modifying or loading large numbers of objects, you obtain best performance by committing only after a significant number of modifications or loads. For example, if you are bulk-loading 10,000 employees into the object server, you probably want to commit after each 500 or 1000 employees have been loaded.

  Certain operations create or modify more objects than are visible to you. Creating or removing indexes can touch a large number of objects, depending upon the length of the path and the size of the collections you are indexing. For this reason, performance can increase dramatically if you commit more frequently in such cases, such as after creating or removing only one, or a very few, indexes.

  Cache sizes are discussed in more detail in the following section. The shared page cache, in particular, is described in “Tuning the Shared Page Cache” on page 12-27.

12.3 Modifying Cache Sizes for Better Performance

GemStone uses four kinds of caches: temporary object space, the Gem private page cache, the Stone private page cache, and the shared page cache. The size of these caches are set by GemStone configuration files and can affect how often the system must access the disk or reclaim storage, both operations that can slow your application significantly.

- The *temporary object space* cache is used to store temporary objects created by your application. Some of these objects may ultimately become permanent and reside on the disk, but probably not all of them. Temporary objects that your application creates merely in order to do its work reside in temporary object space until they are no longer needed, when the Gem's garbage collector reclaims the storage they use. Each Gem session has temporary object space associated with it.
- The *Gem private page cache* is also used to store objects created by your application. While the temporary object space reads and writes memory on a per-object basis, the Gem private page cache reads or writes a page at a time. When you commit objects created by your application, they move first from temporary object space to the Gem private page cache. Your application also moves objects to your Gem private page cache if you overflow your temporary object space. Each Gem session has its own private page cache associated with it.
- The *Stone private page cache* is used to maintain lists of allocated object identifiers and pages for each active Gem process that the Stone is monitoring. The single active Stone process per repository has one Stone private page cache.
- The *shared page cache* is used to hold the *object table*—a structure containing pointers to all the objects in the repository—and copies of the disk pages that hold the objects with which users are presently working. The system administrator must enable the shared page cache in the configuration file for a host. The single active Stone process per repository has one shared page cache per host machine. The shared page cache is automatically enabled for the host machine on which the Stone process is running. Enabling the shared page cache is optional, however, for Gem processes running remotely on other hosts.

Configuration File Cache Size Parameters

Two of the four kinds of caches are associated with Gem processes: the temporary object space and the Gem private page cache. The other two are associated with

the Stone (although both the Gem and the Stone make use of the shared page cache). The size of each kind of cache can be set by means of the appropriate GemStone configuration file. The configuration file for each process sets the size of the associated temporary object space and Gem private page cache. The configuration file for the Stone process sets the size of the shared page cache and the Stone private page cache—only the system administrator is privileged to set these parameters. However, if a Gem session is running remotely and it is the first Gem session on its host, its configuration file sets the size of the shared page cache on that host.

- The amount of memory allocated for the temporary object space is determined by the GEM_TEMPOBJ_CACHE_SIZE parameter. The default size is 585 KB; the minimum is 200 KB; the maximum is 10000 KB.
- The amount of memory allocated for the Gem private page cache is determined by the GEM_PRIVATE_PAGE_CACHE_KB parameter. The default size is 500 KB; the minimum is 496 KB; the maximum is 65536 KB.
- The amount of memory allocated for the Stone private page cache is determined by the STN_PRIVATE_PAGE_CACHE_KB parameter. The default size is 1000 KB; the minimum is 496 KB; the maximum is 65536 KB.
- The amount of memory allocated for the shared page cache is determined by the SHR_PAGE_CACHE_SIZE_KB parameter. The default size is 10000 KB; the minimum is 512 KB; the maximum is limited by the available system memory and the kernel configuration.

Tuning Cache Sizes

The default cache sizes assume a small repository; you will probably benefit from increasing some or all of them when your repository grows to industrial proportions.

Tuning the Temporary Object Space

If your application creates a great many temporary objects, you may need to increase the temporary object space size. If your application makes heavy use of the reduced conflict classes, it is more likely to create large numbers of temporary objects. (See “Classes That Reduce the Chance of Conflict” on page 6-26 for more information about the reduced conflict classes.)

You will probably need to experiment somewhat before you determine the optimum size of the temporary object space for your application. In general, keep the size somewhere between 400 KB and 3 MB. Large applications probably require a temporary object space size of at least 1–1.5 MB.

Tuning the Gem Private Page Cache

The shared page cache can be disabled for remote Gem processes. If the shared page cache is disabled, the Gem reads into its private page cache the entire page on which the object resides whenever it needs to read an object. If it then needs to access another object, GemStone first checks to see if the object is already in its private cache. If it is, no further disk access is necessary. If it is not, it reads another page into its private cache.

When temporary object space overflows, objects are written into the Gem private page cache. Objects in the temporary object space can be reclaimed before being flushed to disk, but objects in the Gem private page cache cannot. Storage for unnecessary or obsolete objects in this cache cannot be reclaimed until the objects are written to the disk. Overflowing temporary object space can therefore can therefore prevent objects from being reclaimed, leading to inefficient storage.

Therefore, if you need to increase either temporary object space or the Gem private page cache, increase the temporary object space first. Because it deals with objects one at a time instead of in page-size increments, and because its storage can be reclaimed, it can deal more effectively with temporary space requirements.

If a typical transaction commits more data than the sum of the default size of the temporary object space and the Gem private page cache, then it is a good idea to set larger values for these caches.

NOTE
If more than one Gem process is running remotely on a single host, we recommend that you enable the shared page cache on that host.

Tuning the Stone Private Page Cache

The Stone uses two pages in its private page cache for each Gem session—one page for the session's object pointers, and one page for the session's page allocation. This information is by its nature ephemeral; none of it is ever needed on disk in the repository, as it changes with each user who logs in or logs out. Nevertheless, if the Stone private page cache overflows, the information is flushed to the shared page cache, where it can waste valuable storage. It is therefore desirable never to overflow the Stone private page cache.

To avoid doing so, configure the Stone private page cache so that it can use at least 16 KB per user (one page is 8 KB). If users are using locks, increase the Stone private page cache by 8 KB for each kind of lock used by each user. For example, one user using read locks requires an additional 8 KB; two users using read locks require an additional 16 KB; one user using read locks and write locks requires an

additional 16 KB; and one user using read locks and two users using both read locks and write locks require an additional 40 KB.

Tuning the Shared Page Cache

Whenever the Gem needs to read an object, it reads into the shared page cache the entire page on which an object resides. If the Gem then needs to access another object, GemStone first checks to see if the object is already in the shared page cache. If it is, no further disk access is necessary. If it is not, it reads another page into the shared page cache.

Ideally, the shared page cache should be much larger than required to hold the entire object table.

The average object is between 50–150 bytes. A 300 MB repository probably contains approximately three million objects. For such a repository, the shared page cache should be at least 12 MB plus additional memory for each user. However, it is undesirable to make the shared page cache large enough to force excessive swapping. Therefore, a rule of thumb to start with is that the shared page cache should use approximately one-third to one-half the RAM on the host machine. This rule assumes, however, that your system is dedicated to running GemStone. If it is not, you may need to decrease the shared page cache size.

In any case, the shared page cache ought to be able to hold at least a quarter of the object table in memory, so for the 300 MB repository posited above, allocate at least 5 MB for the shared page cache.

You will get the best performance when the shared page cache is large enough to hold all the pages on which all currently used objects reside. Therefore, clustering objects appropriately can allow a smaller shared page cache.

For more information about caches, see the *GemStone System Administration Guide*.

12.4 Generating Native Code

You may generate platform native code instead of portable code for frequently used code. Native Code Generation works in the run time environment, based on the size of individual methods and the frequency that the virtual machine calls the method.

By using native code, you can increase the performance of your application, especially is your design is compute-bound. Improvements can be as low as 10% or very dramatic, depending on the type of computation required.

In GemStone, native code is as transparent as possible. As a user, you should not really be able to tell is a method executes using native code or portable code.

Enabling Native Code

You control native code generation with the GemNativeCodeMax and the GemNativeCodeThresold parameters, set in the Gem (or as environment variables). These parameters combine to determine when the virtual machine uses native code and either one can disable native code generation.

Table 12.2 GemNativeCodeMax Values

Negative Integers	No limit on the size of the native code; generate the code regardless of its size
0	Native code size is zero, disabling its generation
1-2048	Set a maximum byte size for the resulting native code

Table 12.3 GemNativeCodeThreshold Values

Negative Integers	Native code is not generated
0	Always generate native code.
Positive integers	Generate native code when the method is invoked this number of times; Default: 4.

Implications of Native Code

Although native code is generally transparent, there are some considerations you should be aware of:

- Significant overhead exists for machines with separate data and instruction caches (especially on multi-processor machines). If you set the GemNativeCodeThresold to a low value for these types of clients, the native code may actually run slower than portable code.
- Size of the native code is larger compared with the portable code (uses more RAM space). If your system is memory limited, native code may adversely affect performance.
- Native code methods are always forced back to portable code for debugging.
- Native code for some platforms may have size limits that the portable code doesn't. For example, a branch may only be 8K bytes away on some systems. If the native code is too large, it may never be converted to native code.
- ProfMonitor results will be inaccurate unless native code is disabled. For efficiency, native code does not accurately maintain ProfMonitor statistics.

In most circumstances, native code generation will provide a measurable improvement in performance. Applications with time-critical or space-critical resources may need to tune the native code generation parameters or even have portions of code rewritten to better utilize this feature.

Chapter 13 Advanced Class Protocol

A class responds to messages defined and stored by its class and its class's superclasses. The classes named Object, Class, and Behavior are superclasses of every class. Although the mechanism involved may be a little confusing, the practical implication is easy to grasp — every class understands the instance messages defined by Object, Class, and Behavior.

You're already familiar with Object's protocol, that enables a class to represent itself as a string, to return the segment it occupies. The class named Class defines the familiar subclass creation message (`subclass:instVarNames:...`) and some protocol for adding and removing class variables and pool dictionaries. Class Behavior defines by far the largest number of useful messages that you may not yet have encountered. This chapter presents a brief overview of Behavior's methods.

Adding and Removing Methods
describes the protocol in class Behavior for adding and removing methods.

Examining a Class's Method Dictionary
describes the protocol in class Behavior for examining the method dictionary of a class.

Examining, Adding, and Removing Categories
describes the protocol in class Behavior for examining, adding, and removing method categories.

Accessing Variable Names and Pool Dictionaries
describes the protocol in class Behavior for accessing the variables and pool dictionaries of a class.

Testing a Class's Storage Format
describes the protocol in class Behavior for testing the storage format of a class.

13.1 Adding and Removing Methods

Class Behavior defines three messages for adding or removing selectors.

Defining Simple Accessing and Updating Methods

Class Behavior provides an easy way to define simple methods for establishing and returning the values of instance variables. For each instance variable named by a symbol in the argument array, the message `compileAccessingMethodsFor:` *arrayOfSymbols* creates one method that sets the instance variable's value and one method that returns it. Each method is named for the instance variable to which it provides access.

For example, this invocation of the method:

```
Animal compileAccessingMethodsFor: #(#name)
```

has the same effect as the Topaz script in Example 13.1:

Example 13.1

```
category: 'Accessing'
method: Animal
name

    ^name

%
category: 'Updating'
method: Animal
name: aName
    name := aName
%
```

All of the methods created in this way are added to the categories named "Accessing" (return the instance variable's value) and "Updating" (set its value).

You can also use `compileAccessingMethodsFor:` to define methods for accessing pool variables and class variables. The only important difference is that to define *class* methods for getting at class variables, you must send `compileAccessingMethodsFor:` to the *class* of the class that defines the class variables of interest. The following code, for example, defines class methods that access the class variables of the class Menagerie:

```
Menagerie class compileAccessingMethodsFor: #( #BandicootTally )
```

This is equivalent to the Topaz script in Example 13.2:

Example 13.2

```
category: 'Accessing'
classmethod: Menagerie
BandicootTally

    ^BandicootTally

%
category: 'Updating'   classmethod: Menagerie
BandicootTally: aNumber

    BandicootTally := aNumber

%
```

Removing Selectors

You can send `removeSelector:` *aSelectorSymbol* to remove any selector and associated method that a class defines. The following example removes the selector `habitat` and the associated method from the class Animal's method dictionary.

```
Animal removeSelector: #habitat
```

To remove a *class* method, send `removeSelector:` to the *class* of the class defining the method. The following example removes one of the class Animal's class methods:

```
Animal class removeSelector:
#newWithName:favoriteFood:habitat:
```

Modifying Classes

You can send `varyingConstraint:` *aClass* to change the constraint on the unnamed (indexed or unordered) instance variables of the receiver. The argument *aClass* must be a kind of class. The receiver, and any subclasses for which a constraint change will result, must be variant.

Two methods defined by Class can help you determine the nature of a class that you are considering modifying; these are `definition` and `hierarchy`. The `definition` method displays the structure of the class, and `hierarchy` shows its place in the class hierarchy.

This example shows how to query GemStone for the definition of Employee:

Example 13.3

```
Employee definition
  Object subclass: 'Employee'
  instVarNames: #( 'name' 'job' 'age' 'address')
  classVars: #( 'subclasses')
  poolDictionaries: #[]
  inDictionary: UserGlobals
  constraints: #[ #[ #name, Name],
                 #[ #job, String],
                 #[ #age, SmallInteger],
                 #[ #address, String] ]
  instancesInvariant: false
  isModifiable: true
```

The following example shows that Employee is a subclass of Object:

Example 13.4

```
Employee hierarchy
  Object
    Employee( 'name' 'job' 'age' 'address')
```

The Basic Compiler Interface

Class Behavior defines the basic method for compiling a new method for a class and adding the method to the class's method dictionary. The programming environments available with Smalltalk provide much more convenient facilities for run-of-the-mill compilation jobs, so you'll probably never need to use this method. It's provided mainly for sophisticated programmers who want to build a custom programming environment, or those who need to generate Smalltalk methods automatically in Smalltalk.

An invocation of the method has this form:

```
aClass compileMethod: sourceString
       dictionaries:  arrayOfSymbolDicts
       category:      aCategoryNameString
```

The first argument, *sourceString*, is the text of the method to be compiled, beginning with the method's selector. The second argument, *arrayOfSymbolDicts*, is an array of SymbolDictionaries to be used in resolving the source code symbols in *sourceString*. Under most circumstances, you will probably use your symbol list for this argument. The final argument simply names the category to which the new method is to be added.

The following code compiles a short method named `habitat` for the class Animal, adding it to the category "Accessing":

Example 13.5

```
Animal compileMethod:
          'habitat
           "Return the value of the receiver''s habitat
            instance variable"

              ^habitat'
       dictionaries: (System myUserProfile symbolList)
       category: #Accessing
```

When you write methods for compilation in this way, remember to double each apostrophe as in the example given above.

If `compileMethod:..` executes successfully, it adds the new method to the receiver and returns nil.

If the source string contains errors, this method returns an array of two-element arrays. The first element of each two-element array is an error number, and the second element is an integer describing where in the source string the compiler detected the error. The *GemStone C Interface Manual* describes the Smalltalk compiler errors by number, all the error messages are listed in *$GEMSTONE/doc/errormessages.txt*.

13.2 Examining a Class's Method Dictionary

Class Behavior defines messages that let you obtain a class's selectors and methods. Methods are available in either source code or compiled form. Other methods let you test for the presence of a selector in a class or in a class's superclass chain. See Table 13.1.

Table 13.1 Method Dictionary Access

allSelectors	Returns an array of symbols, consisting of all of the message selectors that instances of the receiver can understand, including those inherited from superclasses. The symbol for keyword messages concatenates the keywords.
canUnderstand: *aSelector*	Returns true if the receiver can respond to the message indicated by *aSelector*, returns false otherwise. The selector (a string) can be in the method dictionary of the receiver or any of the receiver's superclasses.
compiledMethodAt: *aSelector*	Returns the compiled method associated with the argument *aSelector* (a string). The argument must be a selector in the receiver's method dictionary; if it is not, this method generates an error.
includesSelector: *aString*	Returns true if the receiver defines a method for responding to *aString*.
selectors	Returns an array of symbols, consisting of all of the message selectors defined by the receiver. (Selectors inherited from superclasses are not included.) The symbol for keyword messages concatenates the keywords.
sourceCodeAt: *aSelector*	Returns a string representing the source code for the argument, *aSelector*. If *aSelector* (a string) is not a selector in the receiver's method dictionary, this generates an error.
whichClassIncludesSelector: *aString*	If the selector *aString* is in the receiver's method dictionary, returns the receiver. Otherwise, returns the most immediate superclass of the receiver where *aString* is found as a message selector. Returns nil if the selector is not in the method dictionary of the receiver or any of its superclasses.

Example 13.6 uses `selectors` and `sourceCodeAt:` in a method that produces a listing of the receiver's methods:

Example 13.6

```
classmethod: SomeClass
listMethods
"Returns a string listing the source code for the receiver's class
and instance methods."
| selectorArray outputString |
outputString := String new.
"First, concatenate all instance methods defined by the receiver."
outputString add: '************ Instance Methods *************';
             lf;
             lf.
selectorArray := self selectors.
selectorArray do: [:i | outputString
                       add: (self sourceCodeAt: i);lf;lf].
"Now add the class methods."
outputString add: '************* Class Methods *************';
             lf;lf.
selectorArray := self class selectors.
selectorArray do: [:i | outputString
                        add: (self class sourceCodeAt: i);lf;lf].
^outputString
%
```

Suppose that you defined a subclass of SymbolDictionary called CustomSymbolDictionary and used instances of that class in your symbol list for storing objects used in your programs.

The method in Example 13.7, using `includesSelector:`, would be able to tell you which of the classes in a CustomSymbolDictionary implemented a particular selector.

Example 13.7

```
method: CustomSymbolDictionary
whichClassesImplement: aSelector
"Returns a string describing which classes in the receiver
(a subclass of SymbolDictionary) implement a method whose selector is
aSelector. Distinguishes between class and instance methods."
| outputString newline |
outputString := String new.
self values do: [:aValue | (aValue isKindOf: Class)
        ifTrue: [ (aValue includesSelector: aSelector)
             ifTrue: [ outputString add: aValue name;
                                    add: ' (as instance method)';
                                    lf.
                     ].
          (aValue class includesSelector: aSelector)
             ifTrue: [ outputString add: aValue name;
                                    add: ' (as class method)';
                                    lf.
                     ].
               ].
        ].
^outputString
%
```

13.3 Examining, Adding, and Removing Categories

Class Behavior provides a nice set of tools for dealing with categories and for examining and organizing methods in terms of categories. See Table 13.2.

Table 13.2 Category Manipulation

addCategory: *aString*	Adds *aString* as a method category for the receiver, and returns the receiver. If *aString* is already a method category, generates an error.
categoryNames	Returns an array of symbols. The elements of the array are the receiver's category names (excluding names inherited from superclasses).
moveMethod: *aSelector* toCategory: *categoryName*	Moves the method *aSelector* (a string) from its current category to the specified category (also a string). Returns the receiver. If either *aSelector* or *categoryName* is not in the receiver's method dictionary, or if *aSelector* is already in *categoryName*, this generates an error.
removeCategory: *categoryName*	Removes the specified category and all its methods from the receiver's method dictionary. Returns the receiver. If *categoryName* is not in the receiver's method dictionary, generates an error.
renameCategory: *categoryName* to: *newCategoryName*	Changes the name of the specified category to *newCategoryName* (a string), and returns the receiver. If *categoryName* is not in the receiver's method dictionary, or if *newCategoryName* is already in the receiver's method dictionary, generates an error.
selectorsIn: *categoryName*	Returns an array of all selectors in the specified category. If *categoryName* is not in the receiver's method dictionary, generates an error.

Notice that `removeCategory:` removes not only a category but all of the methods in that category.

Example 13.8 shows how you might move methods to another category before deleting their original category:

Example 13.8

```
Animal addCategory: 'Munging'.
(Animal selectorsIn: 'Accessing') do: [:i | Animal moveMethod: i
                                             toCategory: 'Munging'].
Animal removeCategory: 'Accessing'.
```

The next example demonstrates how you use these methods to lists instance methods in a format that can be read and compiled automatically by Topaz with the *input* function. This partially duplicates the Topaz fileout function.

Example 13.9

```
classmethod: SomeClass
listMethodsByCategory
"Produces a string describing the receiver’s instance methods by
category in FILEOUT format."
| outputString newline className |
outputString := String new.
className := self name.
self categoryNames do:                    "For each category..."
     [:aCatName | outputString add: 'category: ''';
                               add: aCatName;
                               add: '''';
                               lf.
         (self selectorsIn: aCatName) do: "For each selector..."
               [:aSelector |
                  outputString add: 'method: ';
                              add:className; lf;
                              add: (self sourceCodeAt: aSelector);
                              add: newline;
                              add: '%'; lf;lf.
               ].
     ].
^outputString

%
```

Here is how this method behaves if it were defined for class Animal. (The output is truncated.)

Example 13.10

```
Animal listMethodsByCategory
 category: 'Accessing'
 method: Animal
 name: aName

   name := aName
%

 method: Animal
 name

   ^name

%
```

13.4 Accessing Variable Names and Pool Dictionaries

Class Behavior's methods provide access to the names of all of a class's variables (instance, class, class instance, and pool). Two methods access each kind of variable name—one method retrieves the variables defined by the receiver, and one retrieves the names of inherited variables as well. A more general method (`scopeHas:`) asks whether a variable (instance, class, or pool) is defined for a class's methods. See Table 13.3.

Table 13.3 Access to Variable Names

allClassVarNames	Returns an array of the names of class variables addressable by this class. Variables inherited from superclasses are included; contrast with classVarNames.
allInstVarNames	Returns an array of symbols, consisting of the names of all the receiver's instance variables, including those inherited from superclasses. The ordering of the names in the array follows the ordering of the superclass hierarchy; that is, instance variable names inherited from Object are listed first, and those peculiar to the receiver are last.
allSharedPools	Returns an array of pool dictionaries used by this class and its superclasses. Contrast with `sharedPools`.
classVarNames	Returns a (possibly empty) array of class variables defined by this class. Superclasses are not included; contrast with `allClassVarNames`.
instVarNames	Returns an array of symbols naming the instance variables defined by the receiver, but not including those inherited from superclasses. Contrast with `allInstVarNames`.
scopeHas: *aVariableName* ifTrue: *aBlock*	If *aVariableName* (a string) is an instance, class, or pool variable in the receiver or in one of its superclasses, this evaluates the zero-argument block *aBlock* and returns the result of evaluating *aBlock*. Otherwise, returns false.
sharedPools	Returns an array of pool dictionaries used by this class. Superclasses are not included; contrast with `allSharedPools`.

To access class instance variables, send the message instVarNames or allInstVarNames to the *class* of the class you are searching. For example:

```
Animal class instVarNames
```

returns an array of symbols naming the class instance variables defined by the receiver (Animal), not including those inherited from superclasses. The method in Example 13.11 uses this protocol to return a string giving the names and types of the named instance variables stored in elements of the receiver (a constrained collection).

Example 13.11

```
classmethod: BagOfEmployees
schemeAsString
| bagConstraintClass outputString |
outputString := String new.
"Get the constraint on the receiver."
bagConstraintClass := self varyingConstraint.
outputString add: self name;
             add: ' is constrained to hold instances of ';
             add: bagConstraintClass name;
             add: (Character lf).
outputString add: 'Instance variables of ';
             add: bagConstraintClass name;
             add: ' are:';
             add: (Character lf).
self allInstVarnames do:
  [:aVarName | "For each instvarname in the receiver's elements"
      outputString add: aVarname;    "Add the instance var name"
                   add: ' ';
                   add: 'constraint: ';
                   "Add the constraint on that instance var"
                   add: ((bagConstraintClass constraintOn: aVarName)
                      name);
                   add: (Character lf).
       ].
^outputString
%
```

This method does not need not be rewritten when its class is redefined to add or remove instance variables or constraints.

The next method, defined for the class CustomSymbolDictionary (discussed earlier in this chapter), returns a list of all classes named by the receiver that define some name as an instance, class, class instance, or pool variable.

Example 13.12

```
method: CustomSymbolDictionary
listClassesThatReference: aVarName
"Lists the classes in the receiver, a subclass of SymbolDictionary,
that refer to aVarName as an instance, class, class instance, or pool
variable."
| theSharedPools outputString |
outputString := String new.
self valuesDo: [:aValue | "For each value in the receiver's
Associations..."
  (aValue instVarNames includesValue: aVarName)
    ifTrue: [outputString add: aValue name;
                         add: ' defines as instance variable.';
                         lf.
 ].
  (aValue classVarNames includesValue: aVarName)
   ifTrue:
     [outputString add: aValue name;
                  add: ' defines as class variable.';
                  lf.
     ].
theSharedPools:= aValue sharedPools.
theSharedPools do: [:poolDict |
                (poolDict includesKey: aVarName)
                ifTrue:
                   [ outputString add: aValue name;
                                    add: ' defines as pool variable'.
                   ].
            ].
        ].
^outputString
%
```

13.5 Testing a Class's Storage Format

Each class defines or inherits for instances the ability to store information in a certain format. Thus, instances of Array can store information in indexed variables, and instances of Bag can store information in unordered instance variables. Table 13.4 describes Behavior's protocol for testing a class's storage format.

Table 13.4 Storage Format Protocol

Message	Description
format	Returns the value of Behavior's *format* instance variable, a SmallInteger. The following methods provide easier ways to get at the information encoded in *format*. If you really need details about *format*, see the description of class Behavior in the *GemStone Kernel Reference*.
instSize	Returns the number of named instance variables in the receiver.
isBytes	Returns true if instances of the receiver are byte objects. Otherwise returns false.
isIndexable	Returns true if instances of the receiver have indexed variables. Otherwise returns false.
isInvariant	Returns true if instances of the receiver cannot change value. Otherwise returns false.
isNsc	Returns true if instances of the receiver are nonsequenceable collections (IdentityBags or IdentitySets). Otherwise returns false.
isPointers	Returns true if instances of the receiver are pointer objects. Otherwise returns false.

Example 13.13 below uses several of these messages to construct a string describing the storage format of the receiver.

Example 13.13

```
method: Object
describeStorageFormat
"Returns a string describing the receiver's format and the kinds
and numbers  of its instance variables."
| outputString |
outputString := String new.
outputString add: 'The receiver is an instance of ';
             add: self class name;
             add: '.';
             lf;
             add: 'Its storage format is '.
self class isPointers              "Is the receiver pointers?"
  ifTrue: [self class isIndexable
    ifTrue: [ outputString add: 'pointer with indexed vars. ';
                           add: 'The number of indexed vars is ';
                           add: self size asString;
                           lf;
                           add: 'The number of named instvars is ';
                           add: self class instSize asString;
                           add: '.';
                           lf.
            ]
    ifFalse: [ outputString add: 'pointer with no indexed vars.';
                           lf;
                           add: 'The number of named instvars is ';
                           add: self class instSize asString;
                           add: '.';lf.
             ].
      ]
   "If the object has no pointers, then it must be a nonsequenceable
    collection or a byte object."
```

```
   ifFalse:
       [self class isNSC
          ifTrue: [outputString add: 'NSC. ';
                         add:'The number of unordered inst vars is:';
                                add: self size asString;
                                add: '.'.
                  ]
          ifFalse: [outputString add: 'bytes. ';
                          add: 'The number of byte inst vars is: ';
                                add: self size asString;
                                add: '.'.
                ].
        ].
^outputString
%
```

Here's what happens if you define `describeStorageFormat` for class Animal and send it to an instance of Animal:

Example 13.14

```
Animal new describeStorageFormat
 The receiver is an instance of Animal.
 Its storage format is pointer with no indexed vars.
 The number of named instvars is 3.
```

Appendix

A

Basic Smalltalk Syntax

This chapter outlines the syntax for GemStone Smalltalk and introduces some important kinds of Smalltalk objects.

The Smalltalk Class Hierarchy

Every object is an instance of a class, taking its methods and its form of data storage from its class. Defining a class thus creates a kind of template for a whole family of objects that share the same structure and methods. Instances of a class are alike in form and in behavioral repertoire, but independent of one another in the values of the data they contain.

Classes are much like the data types (string, integer, etc.) provided by conventional languages; the most important difference is that classes define actions as well as storage structures. In other words, Algorithms + Data Structures = Classes.

Smalltalk provides a number of predefined classes that are specialized for storing and transforming different kinds of data. Instances of class Float, for example, store floating-point numbers, and class Float provides methods for doing floating-point arithmetic. Floats respond to messages such as +, -, and reciprocal.

Instances of class Array store sequences of objects and respond to messages that read and write array elements at specified indices.

The Smalltalk classes are organized in a treelike hierarchy, with classes providing the most general services nearer the root, and classes providing more specialized functions nearer the leaves of the tree. This organization takes advantage of the fact that a class's structure and methods are automatically conferred on any classes defined as its subclasses. A subclass is said to inherit the properties of its parent and its parent's ancestors.

How to Create a New Class

The following message expression makes a new subclass of class Object, the class at the top of the class hierarchy:

Example 0.1

```
Objectsubclass: 'Animal'
      instVarNames: #( )
      inDictionary: UserGlobals.
```

This subclass creation message establishes a name ('Animal') for the new class and installs the new class in a Dictionary called UserGlobals. The String used for the new class's name must follow the general rule for variable names — that is, it must begin with an alphabetic character and its length must not exceed 64 characters. Installing the class in UserGlobals makes it available for use in the future—you need only write the name Animal in your code to refer to the new class.

Case-Sensitivity

Smalltalk is case-sensitive; that is, names such as “SuperClass,” “superclass,” and “superClass” are treated as unique items by the Smalltalk compiler.

Statements

The basic syntactic unit of a Smalltalk program is the *statement*. A lone statement needs no delimiters; multiple statements are separated by periods:

```
a := 2.
b := 3.
```

In a group of statements to be executed en masse, a period after the last statement is optional.

A statement contains one or more *expressions*, combining them to perform some reasonable unit of work, such as an assignment or retrieval of an object.

Comments

Smalltalk usually treats a string of characters enclosed in quotation marks as a *comment*—a descriptive remark to be ignored during compilation. Here is an example:

```
"This is a comment."
```

A quotation mark does *not* begin a comment in the following cases:

- within another comment. You cannot nest comments.
- within a string literal (see *String Literals* below). Within a Smalltalk string literal, a “comment” becomes part of the string.
- when it immediately follows a dollar sign ($). Smalltalk interprets the first character after a dollar sign as a data object called a character literal (see below).

A comment terminates tokens such as numbers and variable names. For example, Smalltalk would interpret the following as two numbers separated by a space (by itself, an invalid expression):

```
2" this comment acts as a token terminator" 345
```

Expressions

An expression is a sequence of characters that Smalltalk can interpret as a reference to an object. Some references are direct, and some are indirect.

Expressions that name objects directly include both variable names and literals such as numbers and strings. The values of those expressions are the objects they name.

An expression that refers to an object indirectly by specifying a message invocation has the value returned by the message’s receiver. You can use such an expression anywhere you might use an ordinary literal or a variable name. This expression:

```
2 negated
```

has the value (refers to) -2, the object that 2 returns in response to the message `negated`.

The following sections describe the syntax of Smalltalk expressions and tell you something about their behavior.

Kinds of Expressions

A Smalltalk expression can a combination of the following:

- a literal
- a variable name
- an assignment
- a message expression
- an array constructor
- a path
- a block

The following sections discuss each of these kinds of expression in turn.

Literals

A *literal expression* is a representation of some object such as a character or string whose value or structure can be written out explicitly. The five kinds of Smalltalk literals are:

- numbers
- characters
- strings
- symbols
- arrays of literals

Numeric Literals

In Smalltalk, literal numbers look and act much like numbers in other programming languages. Like other Smalltalk objects, numbers receive and respond to messages. Most of those messages are requests for arithmetic operations. In general, Smalltalk numeric expressions do the same things as their counterparts in conventional programming languages. For example:

```
5 + 5
```

returns the sum of 5 and 5.

A literal floating point number must include at least one digit after the decimal point:

```
5.0
```

You can express very large and very small numbers compactly with scientific notation. To raise a number to some exponent, simply append the letter “e” and a numeric exponent to the number’s digits. For example:

```
8.0e2
```

represents 800.0. The number after the e represents an exponent (base 10) to which the number preceding the e is to be raised. The result is always a floating point number. Here are more examples:

```
1e-3    represents   0.001
1.5e0   represents   1.5
```

To represent a number in a nondecimal base literally, write the number’s base (in decimal), followed by the character "#", and then the number itself. Here, for example, is how you would write octal 23 and hexadecimal FF:

```
8#23
16#FF
```

The largest radix available is 36.

Character Literals

A Smalltalk character literal represents one of the symbols of the alphabet. To create a character literal, write a dollar sign ($) followed by the character’s alphabetic symbol. Here are some examples:

```
$b  $B  $4  $?  $$
```

If a nonprinting ASCII character such as a tab or a form feed follows the dollar sign, Smalltalk creates the appropriate internal representation of that character. Smalltalk interprets this statement, for example, as a representation of ASCII character 32:

Example A.2

```
$ . "Creates the character representing a space (ASCII 32)"
```

In this example, the period following the space acted as a statement terminator. If no space had separated the dollar sign from the period, Smalltalk would have interpreted the expression as the character literal representing a period.

String Literals

Literal strings represent sequences of characters. They are instances of the class String, described in Chapter 4, "Collection and Stream Classes." A literal string is a sequence of characters enclosed by single quotation marks. These are literal instances of String:

```
'Intellectual passion drives out sensuality.'
'A difference of taste in jokes is a great strain
 on the affections.'
```

A literal String may contain up to 10,000 characters.

When you want to include apostrophes in a literal string, double them:

```
'You can''t make omelettes without breaking eggs.'
```

Smalltalk faithfully preserves control characters when it compiles literal strings. The following example creates a String containing a line feed (ASCII 10), Smalltalk's end-of-line character:

```
'Control characters such as line feeds
 are significant in literal strings.'
```

As Chapter 4, "Collection and Stream Classes," explains, Strings respond to a variety of text manipulation messages.

Symbol Literals

A literal Symbol is similar to a literal String. It is a sequence of letters, numbers, or an underscore preceded by a pound sign (#). For example:

Example A.3

```
#stuff
#nonsense
#may_24_thisYear
```

Literal Symbols can contain white space (tabs, carriage returns, line feeds, formfeeds, spaces, or similar characters). If they do, they must be both preceded by a pound sign, as described above, and delimited by single quotation marks, as described in the section above entitled "String Literals." For example:

#'Gone With the Wind'

Chapter 4, "Collection and Stream Classes," discusses the behavior of Symbols.

Array Literals

A Smalltalk Array is a simple data structure similar to a FORTRAN array or a LISP list. Arrays can hold objects of any type, and they respond to messages that read and write individual elements or groups of elements.

A literal Array can contain only other literals—Characters, Strings, Symbols, and other literal Arrays. The elements of a literal Array are enclosed in parentheses and preceded by a pound sign (#). White space must separate the elements.

Here is an Array that contains two Strings, a literal Array, and a third String:

```
#('string one' 'string two' #('another' 'Array') 'string three')
```

The following Array contains a String, a Symbol, a Character, and a Number:

```
#('string one' #symbolOne $c 4)
```

Array literals are only one of two flavors of arrays. Array constructors are discussed later in the chapter.

Variables and Variable Names

A variable name is a sequence of characters of either or both cases. A variable name must begin with an alphabetic character or an underscore (“_”) , but it can contain numerals. Spaces are not allowed, and the underscore is the only acceptable punctuation mark. Here are some permissible variable names:

```
zero
relationalOperator
Top10SolidGold
A_good_name_is_better_than_precious_ointment
```

Most Smalltalk programmers begin local variable names with lowercase letters and global variable names with uppercase letters. When a variable name contains several words, Smalltalk programmers usually begin each word with an uppercase letter. You are free to ignore either of these conventions, but remember that Smalltalk is case-sensitive. The following are all different names to Smalltalk:

```
VariableName
variableName
variablename
```

Variable names can contain up to 64 characters.

Declaring Temporary Variables

Like many other languages, Smalltalk requires you to declare new variable names (implicitly or explicitly) before using them. The simplest kind of variable to declare, and one of the most useful in your initial exploration of Smalltalk, is the temporary variable. Temporary variables are so called because they are defined only for one execution of the set of statements in which they are declared.

To declare a temporary variable, you must surround it with vertical bars as in this example:

Example A.4

```
| myTemporaryVariable |
  myTemporaryVariable := 2.
```

You can declare at most 253 temporary variables for a set of statements. Once declared, a variable can name objects of any kind.

To store a variable for later use, or to make its scope global, you must put it in one of GemStone's shared dictionaries that Smalltalk uses for symbol resolution. For example:

Example A.5

```
| myTemporaryVariable |
myTemporaryVariable := 2.
UserGlobals at: #MyPermanentVariable put: myTemporaryVariable.
```

Subsequent references to MyPermanentVariable return the value 2.

Chapter 4, "Collection and Stream Classes," explains Dictionaries and the message `at:put:`. Chapter 3, "Name Resolution and Object Sharing," provides a complete discussion of symbol resolution and discusses other kinds of "implicit declaration" similar to storage in UserGlobals.

Pseudovariables

You can change the objects to which most variable names refer simply by assigning them new objects. However, five Smalltalk variables have values that cannot be changed by assignment; they are therefore called *pseudovariables*. They are:

nil

Refers to an object representing a null value. Variables not assigned another value automatically refer to nil.

true

Refers to the object representing logical truth.

false

Refers to the object representing logical falsity.

self

Refers to the receiver of the message, which differs according to the context.

super

Refers to the receiver of the message, but the method that is invoked is in the superclass of the receiver.

Assignment

Assignment statements in Smalltalk look like assignment statements in many other languages. The following statement assigns the value 2 to the variable MightySmallInteger:

```
MightySmallInteger := 2.
```

The next statement assigns the same String to two different variables (C programmers may notice the similarity to C assignment syntax):

```
nonmodularity := interdependence := 'No man is an island'.
```

Message Expressions

As you know, Smalltalk objects communicate with one another by means of messages. Most of your effort in Smalltalk programming will be spent in writing expressions in which messages are passed between objects. This subsection discusses the syntax of those message expressions.

You have already seen several examples of message expressions:

```
2 + 2
5 + 5
```

In fact, the only Smalltalk code segments you have seen that are not message expressions are literals, variables, and simple assignments:

```
2                             "a literal"
variableName                  "a variable"
MightySmallInteger := 2.      "an assignment"
```

The ubiquity of message-passing is one of the hallmarks of object-oriented programming.

Messages

A message expression consists of:

- an identifier or expression representing the object to receive the message,
- one or more identifiers called selectors that specify the message to be sent, and
- (possibly) one or more arguments that pass information with the message (these are analogous to procedure or function arguments in conventional programming). Arguments can be written as message expressions.

Reserved Selectors

Because GemStone represents selectors internally as symbols, almost any identifier that is legal as a literal symbol is acceptable as a selector. A few selectors, however, have been reserved for the sole use of the Smalltalk kernel classes. Those selectors are:

```
ifTrue:              untilFalse         timesRepeat:
ifFalse:             untilTrue          isNil
ifTrue:ifFalse:      whileFalse:        notNil
ifFalse:ifTrue:      whileTrue:
or:                  to:do:
and:                 to:by:do:
```

Do not create new methods with these selectors. If you do, the Smalltalk compiler will not execute your code.

Optimized Selectors

Certain other selectors are optimized in Smalltalk kernel classes. Redefining an optimized selector in the class for which it is optimized has no effect; the same primitive method will be called and your redefinition will be ignored.

Optimized selectors are listed in Table 0.1:

Table 0.1 Optimized Selectors

Class	Selector
SmallInteger,	+
SmallInteger	-
SmallInteger	*
SmallInteger	>=
SmallInteger	=
any kernel class	==
any kernel class	~~
any kernel class	_class
any kernel class	isKindOf:
any kernel class	_disableProtectedMode
any kernel class	_gsReturnNoResult

Do not redefine these methods in these classes. If you do, the Smalltalk compiler will not execute your code. However, you can redefine them in other classes, such as those comprising your own application.

Messages as Expressions

In the following message expression, the object 2 is the receiver, + is the selector, and 8 is the argument:

```
2 + 8
```

When 2 sees the selector +, it looks up the selector in its private memory and finds instructions to add the argument (8) to itself and to return the result. In other words, the selector + tells the receiver 2 what to do with the argument 8. The object 2 returns another numeric object 10, which can be stored with an assignment:

```
myDecimal := 2 + 8.
```

The selectors that an object understands (that is, the selectors for which instructions are stored in an object's instruction memory or "method dictionary") are determined by the object's class.

Unary Messages

The simplest kind of message consists only of a single identifier called a unary selector. The selector `negated`, which tells a number to return its negative, is representative:

```
7 negated
-7
```

Here are some other unary message expressions:

```
9 reciprocal.  "returns the reciprocal of 9"
myArray last.  "returns the last element of Array myArray"
DateTime now.  "returns the current date and time"
```

Binary Messages

Binary message expressions contain a receiver, a single selector consisting of one or two nonalphanumeric characters, and a single argument. You are already familiar with binary message expressions that perform addition. Here are some other binary message expressions (for now, ignore the details and just notice the form):

```
8 * 8                  "returns 64"
4 < 5                  "returns true"
myObject = yourObject  "returns true if myObject and
                           yourObject have the same value"
```

Keyword Messages

Keyword messages are the most common. Each contains a receiver and up to 15 keyword and argument pairs. In keyword messages, each keyword is a simple identifier ending in a colon.

In the following example, 7 is the receiver, `rem:` is the keyword selector, and 3 is the argument:

```
7 rem: 3 "returns the remainder from the division of 7 by 3"
```

Here is a keyword message expression with two keyword-argument pairs:

Example A.6

```
| arrayOfStrings |
arrayOfStrings := Array new: 4.
arrayOfStrings at: (2 + 1) put: 'Curly'.
arrayOfStrings at: (2 + 1)
 "puts'Curly' at index position 3 in the receiver"
```

In a keyword message, the order of the keyword-argument pairs (`at:`*arg1* `put:`*arg2*) is significant.

Combining Message Expressions

In a previous example, one message expression was nested within another, and parentheses set off the inner expression to make the order of evaluation clear. It happens that the parentheses were optional in that example. However, in Smalltalk as in most other languages, you sometimes need parentheses to force the compiler to interpret complex expressions in the order you prefer.

Combinations of unary messages are quite simple; Smalltalk always groups them from left to right and evaluates them in that order. For example:

```
9 reciprocal negated
```

is evaluated as if it were parenthesized like this:

```
(9 reciprocal) negated
```

That is, the numeric object returned by `9 reciprocal` is sent the message `negated`.

Binary messages are also invariably grouped from left to right. For example, Smalltalk evaluates:

```
2 + 3 * 2
```

as if the expression were parenthesized like this:

```
(2 + 3) * 2
```

This expression returns 10. It may be read: "Take the result of sending + 3 to 2, and send that object the message * 2."

All binary selectors have the same precedence. Only the *sequence* of a string of binary selectors determines their order of evaluation; the identity of the selectors doesn't matter.

However, when you combine unary messages with binary messages, the unary messages take precedence. Consider the following expression, which contains the binary selector + and the unary selector `negated`:

```
2 + 2 negated
0
```

This expression returns the result 0 because the expression `2 negated` executes before the binary message expression 2 + 2. To get the result you may have expected here, you would need to parenthesize the binary expression like this:

```
(2 + 2) negated
-4
```

Finally, binary messages take precedence over keyword messages. For example:

```
myArrayOfNums at: 2 * 2
```

would be interpreted as a reference to myArrayofNums at position 4. To multiply the number at the second position in myArrayOfNums by 2, you would need to use parentheses like this:

```
(myArrayOfNums at: 2) * 2
```

Summary of Precedence Rules

1. Parenthetical expressions are always evaluated first.
2. Unary expressions group left to right, and they are evaluated before binary and keyword expressions.
3. Binary expressions group from left to right, as well, and take precedence over keyword expressions.
4. Smalltalk executes assignments after message expressions.

Cascaded Messages

You will often want to send a series of messages to the same object. By *cascading* the messages, you can avoid having to repeat the name of the receiver for each message. A cascaded message expression consists of the name of the receiver, a message, a semicolon, and any number of subsequent messages separated by semicolons.

For example,

Example A.7

```
| arrayOfPoets |
arrayOfPoets := Array new.
(arrayOfPoets add: 'cummings'; add: 'Byron'; add: 'Rimbaud';
yourself)
```

is a cascaded message expression that equivalent to this series of statements:

Example A.8

```
| arrayOfPoets |
arrayOfPoets := Array new.
arrayOfPoets add: 'cummings'.
arrayOfPoets add: 'Byron'.
arrayOfPoets add: 'Rimbaud'.
arrayOfPoets
```

You can cascade any sequence of messages to an object. And, as always, you are free to replace the receiver's name with an expression whose value is the receiver.

Array Constructors

An array constructor is similar to a literal array, but its elements can be written as nonliteral expressions as well as literals. Smalltalk evaluates the expressions in an Array constructor at run time.

Array constructors look a lot like literal Arrays; the differences are that array constructors are enclosed in brackets and have their elements delimited by commas.

The following example shows an Array constructor whose last element, represented by a message expression, has the value 4.

Example A.9

```
"An Array constructor"
#['string one', #symbolOne , $c , 2+2]
```

Because any valid Smalltalk expression is acceptable as an array constructor element, you are free to use variable names as well as literals and message expressions:

Example A.10

```
| aString aSymbol aCharacter aNumber |
aString := 'string one'.
aSymbol := #symbolOne.
aCharacter := $c.
aNumber := 4.
#[aString, aSymbol, aCharacter, aNumber]
```

The differences in the behavior of array constructors versus literal arrays can be subtle. For example, the literal array:

```
#(123 huh 456)
```

is interpreted as an array of three elements: a SmallInteger, aSymbol, and another SmallInteger. This is true even if you declare the value of *huh* to be a SmallInteger such as 88, as shown below.

Example A.11

```
| huh |
huh := 88.
#[ 123, huh, 456 ]
 [sz:3 cls:  InvariantArray]
   #1 [sz:0 cls:  SmallInteger] 123
   #2 [sz:3 cls:  Symbol]       huh
   #3 [sz:0 cls:  SmallInteger] 456
```

The same declaration used in an array constructor, however, produces an array of three SmallIntegers:

Example A.12

```
| huh |
huh := 88.
#[ 123, huh, 456 ]

[sz:3 cls:  Array]
   #1 [sz:0 cls:  SmallInteger] 123
   #2 [sz:0 cls:  SmallInteger] 88
   #3 [sz:0 cls:  SmallInteger] 456
```

Path Expressions

Most of the syntax described in this chapter so far is standard Smalltalk syntax. However, Smalltalk also includes a syntactic construct called a path. A path is a special kind of expression that returns the value of an instance variable.

A path is an expression that contains the names of one or more instance variables separated by periods; a path returns the value of the last instance variable in the series. The sequence of the names reflects the order of the objects' nesting; the outermost object appears first in a path, and the innermost object appears last. The following path points to the instance variable name, which is contained in the object anEmployee:

```
anEmployee.name
```

The path in this example returns the value of instance variable name within anEmployee.

If the instance variable name contained another instance variable called last, the following expression would return *last*'s value:

```
anEmployee.name.last
```

NOTE:
Use paths only for their intended purposes. Although you can use a path anywhere an expression is acceptable in a Smalltalk program, paths are intended for specifying indexes, formulating queries, and sorting. In other contexts, a path returns its value less efficiently than an equivalent message expression.Paths also violate the encapsulation that is one of the strengths of the object-oriented data model. Using them can circumvent the designer's intention.Finally, paths are not standard Smalltalk syntax. Therefore, programs using them are less portable than other Smalltalk programs.

Returning Values

Previous discussions have spoken of the "value of an expression" or the "object returned by an expression." Whenever a message is sent, the receiver of the message returns an object. You can think of this object as the message expression's value, just as you think of the value computed by a mathematical function as the function's value.

You can use an assignment statement to capture a returned object:

Example A.13

```
| myVariable |
myVariable := 8 + 9.      "assign 17 to myVariable"
myVariable                "return the value of myVariable"
17
```

You can also use the returned object immediately in a surrounding expression:

Example A.14

```
"puts 'Moe' at position 2  in arrayOfStrings"
| arrayOfStrings |
arrayOfStrings := Array new: 4.
(arrayOfStrings at: 1+1 put: 'Moe'; yourself) at: 2
```

And if the message simply adds to a data structure or performs some other operation where no feedback is necessary, you may simply ignore the returned value.

Blocks

A Smalltalk block is an object that contains a sequence of instructions. The sequence of instructions encapsulated by a block can be stored for later use, and executed by simply sending the block the unary message `value`. Blocks find wide use in Smalltalk, especially in building control structures.

A literal block is delimited by brackets and contains one or more Smalltalk expressions separated by periods. Here is a simple block:

```
[3.2 rounded]
```

To execute this block, send it the message `value`.

```
[3.2 rounded] value
3
```

When a block receives the message `value`, it executes the instructions it contains and returns the value of the last expression in the sequence. The block in the following example performs all of the indicated computations and returns 8, the value of the last one.

```
[89*5 . 3+4 . 48/6] value
8
```

You can store a block in a simple variable:

```
| myBlock |
myBlock := [3.2 rounded].
myBlock value.
3
```

or store several blocks in more complex data structures, such as Arrays:

Example A.15

```
| factorialArray |
factorialArray := Array new.
factorialArray at: 1 put: [1];
               at: 2 put: [2 * 1];
               at: 3 put: [3 * 2 * 1];
               at: 4 put: [4 * 3 * 2 * 1].
(factorialArray at: 3) value
6
```

Because a block's value is an ordinary object, you can send messages to the value returned by a block.

Example A.16

```
| myBlock |
myBlock := [4 * 8].
myBlock value / 8
4
```

The value of an empty block is nil.

```
[ ] value
nil
```

Blocks are especially important in building control structures. The following section discusses using blocks in conditional execution.

Blocks with Arguments

You may build blocks that take arguments. To do so, precede each argument name with a colon, insert it at the beginning of the block, and append a vertical bar to separate the arguments from the rest of the block.

Here is a block that takes an argument named *myArg*:

```
[ :myArg | 10 + myArg]
```

As shown here, the colon must *immediately* precede the argument name (*myArg*); white space must not intervene.

To execute a block that takes an argument, send it the keyword message `value:` *anArgument*. For example:

Example A.17

```
| myBlock |
myBlock := [ :myArg | 10 + myArg].
myBlock value: 10.
20
```

The following example creates and executes a block that takes two arguments. Notice the use of the two-keyword message `value:` *aValue* `value:` *anotherValue*.

Example A.18

```
| divider |
divider := [:arg1 :arg2 | arg1 / arg2].
divider value: 4 value: 2
2
```

A block assigns actual parameter values to block variables in the order implied by their positions. In this example, *arg1* takes the value 4 and *arg2* takes the value 2.

Variables used as block arguments are known only within their blocks; that is, a block variable is local to its block. A block variable's value is managed independently of the values of any similarly named instance variables, and Smalltalk discards it after the block finishes execution. Example A.19 illustrates this:

Example A.19

```
| aVariable |
aVariable := 1.
[:aVariable | aVariable ] value: 10.
aVariable
1
```

You cannot assign to a block variable within its block. This code, for example, would elicit a compiler error:

Example A.20

```
"The following expression attempts an invalid assignment
 to a block variable."
[:blockVar | blockVar := blockVar * 2] value: 10
```

Blocks and Conditional Execution

Most computer languages, Smalltalk included, execute program instructions sequentially unless you include special flow-of-control statements. These statements specify that some instructions are to be executed out of order; they enable you to skip some instructions or to repeat a block of instructions. Flow of control statements are usually conditional; they execute the target instructions if, until, or while some condition is met.

Smalltalk flow of control statements rely on blocks because blocks so conveniently encapsulate sequences of instructions. Smalltalk's most important flow of control structures are message expressions that execute a block if or while some object or expression is true or false. Smalltalk also provides a control structure that executes a block a specified number of times.

Conditional Selection

You will often want Smalltalk to execute a block of code only if some condition is true or only if it is false. Smalltalk provides the messages `ifTrue:` *aBlock* and `ifFalse:` *aBlock* for that purpose. Example A.21 contains both:

Example A.21

```
5 = 5 ifTrue: ['yes, five is equal to five'].
yes, five is equal to five
5 > 10 ifFalse: ['no, five is not greater than ten'].
no, five is not greater than ten
```

In the first of these examples, Smalltalk initially evaluates the expression `(5 = 5).` That expression returns the value true (a Boolean), to which Smalltalk then sends the selector `ifTrue:`. true receives this message and looks at itself to verify that it is, indeed, the object true. Because it is, it proceeds to execute the block passed as `ifTrue:`'s argument, and the result is a String.

The receiver of `ifTrue:` or `ifFalse:` must be Boolean; that is, it must be either true or false. In the last pair of examples, the expressions `(5 = 5)` and `(5 > 10)` returned, respectively, true and false, because Smalltalk numbers know how to compute and return those values when they receive messages such as = and >.

As you read more of this manual, you'll learn about other messages that return true or false, and you'll learn more about the objects true and false themselves.

Two-way Conditional Selection

You will often want to direct your program to take one course of action if a condition is met and a different course if it isn't. You could arrange this by sending `ifTrue:` and then `ifFalse:` in sequence to a Boolean (true or false) expression. For example:

Example A.22

```
2 < 5 ifTrue: ['two is less than five'].
two is less than five
2 < 5 ifFalse: ['two is not less than five'].
nil
```

However, Smalltalk lets you express the same instructions more compactly by sending the single message `ifTrue:` *block1* `ifFalse:` *block2* to an expression or object that has a Boolean value. Which of that message's arguments Smalltalk executes depends upon whether the receiver is true or false. In Example A.23 the receiver is true:

Example A.23

```
2 < 5 ifTrue: ['two is less than five']
      ifFalse: ['two is not less than five'].
two is less than five
```

Conditional Repetition

You will also sometimes want to execute a block of instructions repeatedly as long as some condition is true, or as long as it is false. The messages `whileTrue:` *aBlock* and `whileFalse:` *aBlock* give you that ability. Any block that has a Boolean value responds to these messages by executing *aBlock* repeatedly while it (the receiver) is true (`whileTrue:`) or false (`whileFalse:`).

Here is an example that repeatedly adds one to a variable until the variable equals five:

Example A.24

```
| sum |
sum := 0.
[sum = 5] whileFalse: [sum := sum + 1].
sum
5
```

The next example calculates the total payroll of a miserly but egalitarian company that pays each employee the same salary.

Example A.25

```
| totalPayroll numberOfEmployees salariesAdded standardSalary |
totalPayroll := 0.00.
salariesAdded := 0.
numberOfEmployees := 40.
standardSalary := 5000.00.
"Now repeatedly add the standard salary to the total payroll
so long as the number of salaries added is less than the
number of employees"
[salariesAdded < numberOfEmployees] whileTrue:
      [totalPayroll := totalPayroll + standardSalary.
      salariesAdded := salariesAdded + 1].
totalPayroll
2.0E5
```

Blocks also accept two unary conditional repetition messages, `untilTrue` and `untilFalse`. These messages cause a block to execute repeatedly until the block's last statement returns either true (`untilTrue`) or false (`untilFalse`).

The following example, presented before in a different form, uses `untilTrue`:

Example A.26

```
| sum |

sum := 0.
[sum := sum + 1. sum = 5] untilTrue.
sum

5
```

When Smalltalk executes the block initially (by sending it the message value), the block's first statement adds one to the variable sum. The block's second statement asks whether sum is equal to five; since it isn't, that statement returns false, and Smalltalk executes the block again. Smalltalk continues to reevaluate the block as long as the last statement returns false (that is, while sum is not equal to five).

Alert Pascal programmers may have noticed that expressions using `whileTrue:` are similar to Pascal **while** statements, and that expressions incorporating the unary `untilTrue` are analogous to Pascal **repeat-until** constructs.

The descriptions of classes Boolean and Block in the *GemStone Kernel Reference* describe these flow of control messages and others.

Code Formatting

Like Pascal and C, Smalltalk is a free-format language. A space, tab, line feed, form feed, or carriage return affects the meaning of a Smalltalk expression only when it separates two characters that, if adjacent to one another, would form part of a meaningful token.

In general, you are free to use whatever spacing makes your programs most readable. The following are all equivalent:

Example A.27

```
UserGlobals at: #arglebargle put: 123  "Create the symbol"

#['string one',2+2,'string three',$c,9*arglebargle]

#['string one' , 2+2 , 'string three' , $c , 9*arglebargle]

#[ 'string one',
   2 + 2,
   'string three',
   $c,
   9 * arglebargle ]
```

A.1 Smalltalk BNF

This section provides a complete BNF description of GemStone Smalltalk. Here are a few notes about interpreting the grammar:

`A = expr`

This defines the syntactic production ‘A’ in terms of the expression on the right side of the equals sign.

`B = C | D`

The vertical bar ‘|’ defines alternatives. In this case, the production “B” is one of either “C” or “D”.

`C = '<'`

A symbol in accents is a literal symbol.

`D = F G`

A sequence of two or more productions means the productions in the order of their appearance.

`E = [ A ]`

Brackets indicate optional productions.

`F = { B }`

Braces indicate zero or more occurrences of the productions contained within.

`G = A | (B|C)`

Parentheses can be used to remove ambiguity.

In the Smalltalksyntactic productions in Figure A.1, white space is allowed between tokens.

Figure A.1 Smalltalk BNF

```
AExpression = Primary [ AMessage { ';' ACascadeMessage } ]
ABinaryMessage = ABinarySelector Primary [ UnaryMessages ]
ABinaryMessages = ABinaryMessage { ABinaryMessage }
ACascadeMessage = UnaryMessage | ABinaryMessage | AKeyWordMessage
AKeyWordMessage = AKeyWordPart { AKeyWordPart }
AKeyWordPart = KeyWord Primary UnaryMessages { ABinaryMessage }
AMessage = [UnaryMessages] [ABinaryMessages] [AKeywordMessage]
Array = '(' { ArrayItem } ')'
ArrayBuilder = '#[' [ AExpression { ',' AExpression } ] ']'
ArrayLiteral = '#' Array
ArrayItem = Number | Symbol | SymbolLiteral | StringLiteral |
            CharacterLiteral | Array | ArrayLiteral
Assignment = VariableName ':=' Statement
BinaryMessage = BinarySelector Primary [ UnaryMessages ]
BinaryMessages = BinaryMessage { BinaryMessage }
BinaryPattern = BinarySelector VariableName
Block = '[' [ BlockParameters ] [ Temporaries ] Statements ']'
BlockParameters = { Parameter } '|'
CascadeMessage = UnaryMessage | BinaryMessage | KeyWordMessage
Expression = Primary [ Message { ';' CascadeMessage } ]
KeyWordMessage = KeyWordPart { KeyWordPart }
KeyWordPart = KeyWord Primary UnaryMessages { BinaryMessage }
KeyWordPattern = KeyWord VariableName {KeyWord VariableName}
Literal = Number | NegNumber | StringLiteral | CharacterLiteral |
          SymbolLiteral | ArrayLiteral | SpecialLiteral
Message = [UnaryMessages] [BinaryMessages] [KeyWordMessage]
MessagePattern = UnaryPattern | BinaryPattern | KeyWordPattern
Method = MessagePattern [ Primitive ] MethodBody
MethodBody = [ Temporaries ] [ Statements ]
NegNumber = '-' Number
Operand = Path | Literal | Identifier
Operator = '=' | '==' | '<' | '>' | '<=' | '>=' | '~=' | '~~'
ParenStment = '(' Statement ')'
Predicate = ( AnyTerm | ParenTerm ) { '&' Term }
Primary = ArrayBuilder | Literal | Path | Block | SelectionBlock|ParenStment
Primitive = '<' 'primitive:' Digits '>'
SelectionBlock = '{' Parameter } '|' Predicate '}'
Statement = Assignment | Expression
Statements = { Statement '.' } [ ['^'] Statement ['.'] ]
Temporaries = '|' { VariableName } '|'
ParenTerm = '(' AnyTerm ')'
Term = ParenTerm | Operand
AnyTerm = Operand [ Operator Operand ]
UnaryMessage = Identifier
UnaryMessages = { UnaryMessage }
UnaryPattern = Identifier
```

Smalltalk lexical tokens are shown in Figure A.2. No white space is allowed within lexical tokens.

Figure A.2 Smalltalk Lexical Tokens

```
ABinarySelector = any BinarySelector except comma
BinaryExponent = ( 'e' | 'E' | 'd' | 'D' ) ['-' | '+'] Digits
BinarySelector = ( SelectorCharacter [SelectorCharacter]) |
                 ( '-' [ SelectorCharacter ] )
Character = Any Ascii character with ordinal value 0..255
CharacterLiteral = '$' Character
Comment = '"' { Character } '"'
DecimalExponent = ( 'f' | 'F' ) ['-' | '+'] Digits
Digit = '0' | '1' | '2' | ... | '9'
Digits = Digit {Digit}
Exponent = BinaryExponent | DecimalExponent
FractionalPart = '.' Digits [Exponent]
Identifier = Letter { Letter | Digit }
KeyWord = Identifier ':'
Letter = 'A' | 'B' | ... | 'Z' | 'a' | 'b' | ... | 'z' | '_'
Number = RadixedLiteral | NumericLiteral
Numeric = Digit | 'A' | 'B' | ... | 'Z' | 'a' | 'b' | ... | 'z'
NumericLiteral = Digits ( [FractionalPart] | [Exponent] )
Numerics = Numeric { Numeric }
Parameter = ':' VariableName
Path = Identifier '.' PathIdentifier { '.' PathIdentifier }
PathIdentifier  =  Identifier |  '*'
RadixedLiteral = Digits ( '#' | 'r' ) ['-'] Numerics
SelectorCharacter = '+' | '\' | '*' | '~' | '<' | '>' | '='
            | '|' | '/' | '&' | '@' | '%' | ',' | '?' | '!'
SpecialLiteral = 'true' | 'false' | 'nil'
StringLiteral = "'" { Character | "''" } "'"
Symbol = Identifier | BinarySelector | ( Keyword { Keyword } )
SymbolLiteral = '#' ( Symbol | StringLiteral )
VariableName = Identifier
```

Appendix

B GemStone Error Messages

All Gem Stone errors reside in the array GemStoneError. A Symbol Dictionary "ErrorSymbols" maps mnemonic symbols to the error numbers.

This appendix describes the various types of Gem Stone errors:

- **Compiler errors** report incorrect syntax in a Smalltalk program. You can determine if a GSError is a compiler error by sending it the message **`isCompilerError`**.
- **Interpreter errors** interrupt execution (which can sometimes be restarted). You can determine if a GSError is a compiler error by sending it the message **`isInterpreterError`**.
- **Runtime errors** are detected by the Smalltalk interpreter or by the underlying virtual machine. If execution is broken between byte codes, your application can send its session a message to proceed, if it makes sense to do so. Because an arbitrary amount of Smalltalk code could have executed before the error occurred, your GemStone session could be left in an inconsistent state. It is your responsibility to determine whether the transaction can continue or if you must abort and restart.
- **Aborting errors** indicate that the user's transaction has been aborted. You can send **`isAbortingError`** to test for aborting errors.

- **Fatal errors** indicate that the GemStone system is unavailable, or that a software or hardware malfunction has occurred. (The system could be unavailable because of operating system problems, a login failure, or because the system administrator has disabled GemStone.) You can send **`isFatalError`** to test for fatal errors.

B.1 Description of a Gem Error

All errors except compiler errors are described by the error number and from 0 to 10 optional arguments. Each error below has a comment describing the error and the arguments to the error.

Compiler Errors

Category OOP_COMPILER_ERROR_CAT — numbered from 1001 to 1999

Compiler errors are reported differently than other errors. Rather than having a single error and zero or more arguments, all compiler errors are reported as error COMPILER_ERR_STDB (defined below). This error has at least one argument, which is an array of error descriptors. Each error descriptor is an array of three elements:

- an error number (numbered from 002 to 999 see below)
- an offset into the source code string pointing to where the error was detected
- string describing the error

If the compilation error occurred during automatic recompilation of methods for class modification, the error has 5 additional arguments:

Arg. 2: The source string of the compilation in error.

Arg. 3: The receiver of the recompilation method.

Arg. 4: The category of method containing the error.

Arg. 5: The symbolList used by the recompilation.

Arg. 6: The selector of the method containing the error.
See **`Behavior | recompileAllMethodsInContext:`** for more details.

Error	Error Symbol	Error Message
1001	#compilerErrStDB	Compilation errors -- parameter 1 is error descriptor
1002	#StDBErrMaxPath	Path too long
1003	#StDBErrArrayTooLarge	Array too large
1004	#StDBErrEofInStrLit	Missing end of literal mark (prime)
1005	#StDBErrEofInComment	Missing end of comment (")
1006	#StDBErrEofInChrLit	Invalid or missing character
1007	#StDBErrUnexpectedColon	Invalid colon
1008	#StDBErrUnexpectedPound	Invalid pound sign
1009	#StDBErrBadChr	Illegal symbol
1010	#StDBErrLitTooBig	String literal too big
1011	#StDBErrBadRadix	Illegal radix
1012	#StDBErrBadChrInRadixNum	Illegal character in number
1013	#StDBErrExpectedRightParen	Expected a right parenthesis
1014	#StDBErrExpectedPrimary	Expected a primary expression
1015	#StDBErrExpectedVar	Expected a variable name
1016	#StDBErrExpectedSelector	Missing or bad message pattern
1017	#StDBErrMaxArg	Too many arguments
1018	#StDBErrExpectedEof	Characters found after end of method
1019	#StDBErrExpectedStatement	Illegal character
1020	#StDBErrExpectedPrim	Expected keyword "primitive"
1021	#StDBErrExpectedPrimNum	Expected an integer
1022	#StDBErrBadPrimNum	Illegal primitive number
1023	#StDBErrExpectedRightBrace	Expected the end of the SelectBlock (})
1024	#StDBErrMaxArgsTemps	Too many arguments and temporaries
1025	#StDBErrExpectedVerticalBar	Missing end of temporaries mark (|)

1026	#StDBErrExpectedMsgPattern	Invalid message pattern
1027	#StDBErrExpectedGt	Missing end of primitive mark (>)
1028	#StDBErrBadFlt	Illegal exponent
1029	#StDBErrExpectedAssignable	Expected a variable
1030	#StDBErrAlreadyDefined	Variable has already been declared
1031	#StDBErrNotDefined	Undefined symbol
1032	#StDBErrPredicateTooBig	Predicate too complex
1033	#StDBErrBlocksTooDeep	Blocks can only be nested 31 levels
1034	#StDBErrUnexpectedToken	Unexpected token
1035	#StDBErrExpectedRightBracket	Expected a right bracket (])
1036	#StDBErrStackTooBig	Method too complex
1037	#StDBErrGaijiNotSupported	A Gaiji character was encountered in the source string
1038	#StDBErrCodeTooBig	Method too large
1039	#StDBErrMaxLits	Too many literals
1040	#StDBErrMaxSelectors	Too many selectors
1041	#StDBErrPrimNotAllowed	Only SystemUser may compile a primitive or protected method.
1042	#StDBErrExpectedBoolOrExpr	The object was not true, false or a non-literal expression.
1043	#StDBErrExpectedBlockOrExpr	The object was not a block or a non-literal expression.
1044	#StDBErrExpectedIntOrExpr	The object was not a kind of Integer or a non-literal expression.
1045	#StDBErrNotPosIntOrExpr	The object was not a positive kind of Integer or a non-literal expression.

1046	#StDBErrDisallowedSelector	You may not compile a method for this selector.
1047	#StDBErrBadNumOfArgs	The block has the wrong number of arguments for this selector.
1048	#StDBErrLiteralInvariant	Attempt to modify an invariant literal.
1049	#StDBErrExpectedOperand	An operand was missing from the given SelectBlock term.
1050	#StDBErrBadSelectOperator	An unacceptable operator was given in the SelectBlock term. The operator must be one of <, >, <=, >=, =, ~=, ==, or ~~.
1051	#StDBErrExpectedSelectTerm	The given SelectBlock must contain a Boolean expression.
1052	#StDBErrTermsNotParen	The conjoined terms of the given SelectBlock were not parenthesized.
1053	#StDBErrBadNumOfSelectArgs	A SelectBlock was encountered that does not have one argument.
1054	#StDBErrSourceNotEUCFormat	The bytes of the source string are not in EUC format.
1055	#StDBErrTooManyBlocks	The maximum number of blocks in amethod is 65536.
1056	#StDBErrMaxArgs	The maximum number of arguments to a method or block is 255.
1057	#StDBErrCodeGenLogic	Internal logic error in compiler:
1058	#StDBErrFirstTermCannotBeSetValued	The first term of a path in a SelectBlock cannot indicate search over a set-valued instance variable (i.e. cannot be *).
1059	#StDBErrIllegalProtectedMethod	The specified primitive may not be a protected method.

1060	#StDBMissingProtectedToken	Method requires either <protected> or <unprotected> directive.
1061	#StDBSuperNotAllowed	Reference to super not allowed in instance method for Object.
1062	#StDBUnusedTemp	Unused method or block temporary.
1063	#StDBDbStrOddSize	Corrupt source string, a DoubleByteString has odd basic size.
2001	#rtErrInvalidTransMode	<arg1> is not recognized as a valid transactionMode.
2002	#rtErrBadSymbolList	The user's symbol list is not a kind of Array containing objects that are a kind of SymbolDictionary.
2003	#objErrBadOffsetIncomplete	An indexable object or NSC <arg1> was referenced with an index <arg2> that was out of range.
2004	#rtErrBadSubscript	A subscript <arg2> that was out of range or not an Integer was used to index the object <arg1>.
2005	#gciErrBadNumMsgParts	GciSendMsg was called with an inconsistent number of message parts.
2006	#rtErrAuthArraySize	An attempt was made to change Segment authorization with an array <arg1> which should be of size 5.
2007	#rtErrShouldNotImplement	A method was invoked that has been specifically disallowed in a subclass. Receiver: <arg1>. Selector: <arg2>.
2008	#rtErrSubclassResponsibility	A method has been invoked in the abstract superclass <arg1> that was designed to have been overridden in a concrete subclass. Selector: <arg2>.

2009	#objErrClassVariant	An attempt was made to create an instance of the modifiable class <arg1> . Send "immediateInvariant" to the class to terminate class modification and allow instance creation.
2010	#rtErrDoesNotUnderstand	No method was found for the selector <arg2> when sent to <arg1> with arguments contained in <arg3>.
2011	#objErrNotSegment	An attempt was made to use the object <arg1> as a segment object.
2012	#objErrNotIndexable	An attempt was made to index the object <arg1> that is not indexable.
2013	#rtErrCantPerform	<arg1> cannot perform the selector <arg2> with the arguments in <arg3>. Perform may have been attempted with wrong number of args.
2014	#classErrSubclassDisallowed	Cannot create a subclass of the class <arg1>.
2015	#objErrNotInColl	The object <arg2> was not found in the collection <arg1>.
2016	#repErrMaxExtents	An attempt was made to create a new extent when the logical Repository already has the maximum number of extents (<arg1>) attached.
2017	#rtErrBadPattern	<arg1> is an illegal pattern for string comparisons.
2018	#rtErrBadBlockArgCount	An attempt was made to evaluate the block or method <arg1> with <arg3> arguments when <arg2> were expected.
2019	#objErrCollectionEmpty	An attempt was made to access elements of the empty collection <arg1>.
2020	#rtErrArgNotChr	An attempt was made to store the object <arg1> into a string.
2021	#rtErrKeyNotFound	A reference into the dictionary <arg1> using the non-existent key <arg2> was made.

2022	#rtErrBadDateTimeArgs	Invalid arguments given to DateTime instance creation.
2023	#genericKernelError	Error, <arg1>
2024	#rtErrNoSuchBp	The specified breakpoint does not exist.
2025	#repErrCantCreateFile	The system was unable to create the file <arg1>.
2026	#numErrIntDivisionByZero	An attempt was made to divide <arg1> by zero.
2027	#rtErrSpecialOrNotCommitted	An attempt was made to add the special or uncommitted object <arg1> to the NotifySet.
2028	#repErrPreGrowFailure	The extent <arg1> could not be created because an attempt to pre-grow the file failed for disk capacity reasons.
2029	#rtErrBeginTrans	An attempt was made to begin a new transaction when already in a transaction.
2030	#rtErrPrimOutsideTrans	An attempt was made to execute a primitive that is not allowed when not inside of a transaction. Examples are: commit, backup and restore.
2031	#objErrInvariant	An attempt was made to change the invariant object <arg1>.
2032	#classErrMethCatExists	An attempt was made to create the method category <arg2> which already exists. Class: <arg1>.
2033	#classErrSelectorNotFound	A reference was made to the selector <arg2> which could not be found in the class <arg1> method dictionary.
2034	#lockErrRemove	The user is not allowed to remove a lock on an object <arg1> that they do not have a lock on.
2035	#classErrMethCatNotFound	In searching the class <arg1> the category name <arg2> was not found.
2036	#classErrByteObjInstVars	An attempt was made to create a byte subclass with instance variables. Superclass: <arg1>.

2037	#classErrConstraintNotClass	The constraint <arg1> was specified incorrectly for subclass creation. For IdentityBags (NSCs), a constraint must be specified as a class; for all other classes, as an array of pairs.
2038	#classErrInvariantSuperClass	An attempt was made to create a variant subclass of an invariant class.
2039	#classErrNscNotIndexable	An attempt was made to create an indexable subclass of the NSC class <arg1>.
2040	#repErrExtentNotMounted	The extent with filename or extentId: <arg1> was not part of the logical Repository.
2042	#classErrNscInstVars	An attempt was made to create an NSC subclass <arg1> with instance variables.
2043	#classErrClassVarNameExists	An attempt was made to create the new class variable <arg2> with the same name as an existing class variable in class <arg1>.
2044	#classErrPoolDictExists	An attempt was made to add the dictionary <arg2> to a shared pool of class <arg1> in which it was already a member.
2045	#classErrPoolDictNotFound	An attempt was made to remove the dictionary <arg2> from the shared pool of class <arg1> in which it was not a member.
2046	#clampErrNoSuchInstvar	During clamp compilation for class <arg1> an instance variable clamp was encountered for non-existent instance variable <arg2>.
2047	#clampErrNotAClass	An object <arg1> was specified for instance variable clamping that was not a class object.
2048	#clampErrNotAClampspec	In an attempt to perform clamped object traversal, the specified object <arg1> was not a ClampSpecification object.
2049	#clampErrBadArg	The object <arg1> has an implementation or size not allowed in clamp specifications.
2050	#repErrReplicateOnline	The given extent already is being replicated by <arg1>.

2051	#repErrBadExtentSize	The given maximum extent size (<arg1>) is smaller than the minimum size (<arg2>) allowed for an extent.
2052	#repErrCantOpenFile	The file <arg1> could not be opened.
2053	#rtErrNoSuchInstVar	The instance variable <arg2> was not found in evaluating a path expression for the object <arg1>.
2054	#rtErrTagNotAllowed	An attempt was made to put a tag on the special object <arg1> which is not allowed.
2055	#rtErrBadTagNum	The tag index <arg2> requested for object <arg1> is not allowed. The legal tag indexes are 1 and 2.
2056	#segErrMaxSegGroups	An attempt was made to add the group <arg2> to the segment <arg1>, which already recognizes four groups.
2057	#segErrBadAuthKind	An attempt was made to change the authorization for the segment <arg1> to the unrecognized value <arg2>.
2058	#rtUnresolvedFwdRefs	Commit failed. GciCreate/GciStore have left unsatisfied forward references to the object <arg1>. Create the object with GciCreate and retry commit.
2059	#rtErrStackLimit	Smalltalk DB execution stack overflow.
2060	#rtErrArgNotPositive	<arg1> was found where a positive numeric value was expected.
2061	#rtErrArgOutOfRange	The following argument is too large or out of range: <arg1>
2062	#rtErrCannotChgConstraint	A constraint cannot be changed in a Dictionary that is not empty.
2063	#rtErrNoMessage	There is no error message for the error <arg1> in the SymbolDictionary <arg2>.
2064	#numErrArgNotChr	An attempt was made to coerce the integer <arg1> into a <arg2>, but its value was not in range.

2065	#numErrArgNotFltException	An unrecognized float exception <arg1> was specified.
2066	#numErrFltException	A floating point exception of type <arg1> was raised on the operation <arg2>. The default result is <arg3>. The first argument is <arg4>.
2067	#numErrArgNotRoundingMode	An unrecognized float rounding mode <arg1> was specified.
2068	#segErrCantMoveObj	An attempt was made to change the segment of the special object <arg1>.
2069	#rtErrExceptionNotLinked	An attempt was made to unlink an exception <arg1> that is not installed.
2070	#numErrArgNotFltStatus	An invalid float status <arg1> was specified.
2071	#lockErrUndefinedLock	A request to lock object <arg1> using an invalid kind <arg2>.
2072	#rtErrBadDictConstraint	Only Association or one of its subclasses can be used as a constraint in <arg1>.
2073	#lockErrIncomplete	One or more of the locks you requested was denied or is dirty.
2074	#lockErrObjHasChanged	A request to lock object <arg1> was granted, but the object has been read or written since you started this transaction.
2075	#lockErrDenied	A request to lock object <arg1> with lock kind <arg2> was denied.
2076	#rtErrMaxClusterId	A request to create a new clusterbucket exceeded the maximum size allowed for a clusterId = <arg1>.
2077	#rtErrBadErr	An attempt to signal error <arg1> in dictionary <arg3> was made, but the signal number <arg1> is less than one.
2078	#rtErrUserIdAlreadyExists	An attempt was made to add a UserProfile to the UserProfileSet <arg1> which already has an entry with the same userId: <arg2>.

2079	#rtErrCantReturn	Smalltalk DB execution could not return from the current activation. Home context of block to return from may no longer be active.
2080	#rtErrCantChangeClass	An illegal attempt was made to change the class of the object <arg1> to class <arg2>.
2081	#rtErrCantBecomeSpecial	An attempt was made to use become on the object <arg1> that has a special implementation.
2082	#rtErrGarbageCollect	Attempt to run markForCollection when not the only user on the system and the concurrency mode is set for NO_RW_CHECKS_NO_READ_SET.
2083	#rtErrPrimNotFound	Primitive number <arg1> does not exist in the virtual machine.
2084	#rtErrNoInstVars	An attempt was made to directly access the instance variables of the object <arg1> but the object has no instance variables.
2085	#rtErrExpectedBoolean	Expected <arg1> to be a Boolean.
2086	#rtErrDirtyObjsNeedsInit	GciDirtyObjsInit must be executed at least once before GciDirtySaveObjs.
2087	#rtErrCantChangePassword	An illegal attempt was made to change the password of <arg1>, which is not the UserProfile of the current session.
2088	#rtErrNewStackLimit	The value <arg1> specified for the new stack limit is too large or is smaller than the current size of the execution stack.
2089	#rtErrBadCopyFromTo	An index range was specified for a sequenceable collection with the starting index <arg1> greater than the ending index <arg2>.
2090	#rtErrNilKey	An illegal attempt was made to store nil as a key in the dictionary <arg1>.

2091	#rtErrCantBecomeBothIdx	Both the receiver and argument of become participate in indexes. Become is not allowed because they are not of the same class. <arg1> , <arg2>
2092	#rtErrNoProcessToContinue	The Process <arg1> to continue from is invalid.
2093	#rtErrBadStreamPosition	An attempt was made to set the Stream <arg1> to position <arg2> beyond the limits of the collection.
2094	#rtErrBadArgKind	The object <arg1> was not of the expected class <arg2>.
2095	#classErrClassVarNotFound	An attempt was made to remove the non-existent class variable <arg2> from the class <arg1>.
2096	#assocErrNoElementsDetected	detect: was sent to a collection, but no elements of the collection <arg1> satisfied the block <arg2>.
2097	#classErrNotAVar	The symbol <arg2> was not resolvable as a variable within the class <arg1>.
2098	#segErrTooManyGroups	The collection of groups <arg1> was specified that held more than four members.
2099	#rtErrExpectedByteValue	Byte objects store values from 0 to 255, not <arg1>.
2100	#classErrBadFormat	The representation for new class <arg1> was illegal.
2101	#objErrDoesNotExist	The object with object id <arg1> does not exist.
2102	#objErrNotOopKind	The object <arg1> is not implemented as a pointer object.
2103	#objErrNotByteKind	The object <arg1> is not implemented as a byte object.
2104	#objErrNotNscKind	The object <arg1> is not implemented as an NSC object.

2105	#objErrAlreadyExists	Attempt to create an object with object identifer <arg2>, which already exists as object <arg1>.
2106	#objErrOopNotAllocated	Attempt to store a forward reference using the object identifier <arg1> which has not been allocated to this session.
2107	#objErrConstraintViolation	Attempt to store <arg2> of class <arg4> into an instance variable of <arg1> constrained to be <arg3>.
2108	#rtErrExpectedClass	The object <arg1> was expected to be a class but was not.
2109	#objClassNotOopKind	The class <arg1>, used as argument to GciCreateOopObj does not specify pointer object format.
2110	#objErrBadOffset	The object <arg1> was indexed using structural access with the index <arg3> that was out of range. The maximum index is <arg2>.
2111	#objErrCantCreateInstance	Creating an instance of class <arg1> is not allowed.
2112	#objClassNotByteKind	The class <arg1>, used as argument to GciCreateByteObj does not specify byte object format.
2113	#lockErrArgSize	An argument to a locking primitive was too large.
2114	#objErrNotSpecialKind	The object <arg1> is not implemented as a special object.
2115	#authErrSegRead	An attempt was made to read the object with id <arg1> in segment <arg2> with insufficient authorization.
2116	#authErrSegWrite	An attempt was made to modify the object <arg1> in segment <arg2> with insufficient authorization.
2117	#objErrNotOopOrNsc	An operation was attempted on the object <arg1> that is legal only on pointer or NSC objects.

2118	#rtErrObsolete	<arg1> cannot respond to the selector <arg2> because its class is obsolete.
2119	#rtErrCantBecomeOneIdx	Become is not allowed because the object <arg1> participates in an index and the object <arg2> has a different format.
2120	#objErrNotFlt	The object <arg1> is not a float.
2121	#rtErrCantBecomeClassKind	Become is not allowed because the object <arg1> is a kind of <arg2>
2122	#classErrByteSubclass	You may not create a byte subclass of <arg1>.
2123	#repErrBadBkupSwizzle	An invalid swizzle transform was detected in the backup file long transform: <arg1> short transform: <arg2>
2124	#repErrCantCreateRepos	The Repository <arg1> could not be created.
2125	#repErrBadFileSpec	Invalid file specification.
2126	#repErrFileAlreadyExists	The file <arg1> already exists.
2127	#rtErrDuplicateKey	The key <arg2> already exists in dictionary <arg1>
2128	#assocErrBadComparison	The evaluation of a SelectBlock resulted in an illegal comparison being performed.
2129	#repErrIncompatibleRepos	The Repository version does not match the version of GemStone currently being run.
2130	#assocErrClassModifiable	Index creation failed because the class <arg1> is modifiable. Send "immediateInvariant" to the class to enable index creation.
2131	#classErrConstrInher	The constraint <arg1> is not a subclass of the inherited constraint <arg2> for offset <arg3>.
2132	#classErrBadConstraint	The constraint field was invalid in subclass creation.
2133	#repErrBadBkupVersion	The backup file is incompatible with this version of GemStone. Backup file version is: <arg1>

2134	#objErrBadFetchOffset	Structural access retrieval from object <arg1> used index <arg3> that was out of range. Size of the object is <arg2>.
2135	#rtErrCantBecomeIndexedNsc	Become is not allowed because the object <arg1> is a kind of Bag and currently has indexes
2136	#rtErrNoIndexForPath	An index with the path <arg2> was not found for the Nsc.
2137	#objClassNotOopOrNscKind	The class <arg1>, used as argument to GciCreateOopObj does not specify pointer or nsc object format.
2138	#rtMaxRecursion	Too many levels of recursion from useractions to GemStone Smalltalk or within object manager.
2139	#rtErrBadSession	A non-existent session was specified.
2140	#rtErrNotOnlyUser	An operation that requires exclusive use of the system was attempted when <arg1> users were logged in.
2141	#objErrMaxSize	An attempt was made to extend an object to size <arg2> when the maximum legal size is <arg3>, object:
2142	#rtErrInvalidMethod	The method <arg1> cannot be executed until it has been upgraded to a 4.0 version.
2143	#repErrMaxOnlineRepos	An attempt was made to attach more repositories than can be on-line at one time.
2144	#rtErrRcQueueEntriesFound	In attempting to reset the maxSessionId for the queue to a smaller value, an entry was found that was added by a session whose sessionId is larger than the value currently being set. Remove and save all entries in the queue. Then changeMaxSessionId and add the saved entries back into the queue.

2145	#rtErrFirstPathTermSetValued	The first term in an Nsc index path expression cannot indicate a set-valued instance variable (i.e. cannot be an asterisk).
2146	#gciErrParentSession	The attempted GemBuilder for C operation is illegal in a user action when applied to a session owned by a parent GemBuilder for C or GemStone Smalltalk scope.
2147	#gciErrAlreadyLoggedIn	An attempt was made to log in after the session was already established.
2148	#rtErrInvalidConstraintForMigration	The migration of <arg1> could not occur because instance variable <arg2> with value <arg3> is not an instance of <arg4>
2149	#classErrBadIdentifier	An illegal identifier <arg1> was used to name an instance variable or class.
2150	#classErrConstrPrivateIv	Illegal attempt to constrain the private instanceVariable <arg1>
2151	#rtErrNoPriv	An attempt was made to do a privileged operation for which no privilege had been granted.
2152	#rtErrInvalidBtreeReadStream	The btree read stream is invalid (possibly due to modifications to objects referenced by the stream).
2153	#rtErrDecrNotAllowed	The RcPositiveCounter with value <arg3> cannot be decremented by <arg2>.
2154	#repErrReposNotAttached	The Repository <arg1> is not attached.
2155	#repErrReposNeedsRecovery	The Repository <arg1> could not be attached because it was left in an inconsistent state.
2156	#repErrReplicateNotMounted	The replicate <arg1> is not mounted.

2157	#repErrReposRead	A read error was detected when reading from the Repository <arg1> on page number <arg2>.
2158	#repErrReposWrite	A read error was detected when writing to the Repository <arg1> on page number <arg2>.
2159	#rtErrInvalidElementConstraintForMigration	The migration of <arg1> could not occur because one of its elements <arg2> is not a kind of <arg3> required in the destination class.
2160	#rtErrSelectiveAbort	The selectiveAbort primitive was attempted on an object (<arg1>) that is involved in an index.
2161	#objErrDateTimeOutOfRange	The DateTime passed is either out of range for a time_t, or the julianSeconds field is not between 0 and 86399 inclusive.
2162	#objErrLongNotSmallInt	The integer passed to GciLongToOop was outside the range of SmallIntegers.
2163	#objErrNotLong	GciLongToOop was passed an object which was not a SmallInteger.
2164	#objErrNotChr	GciOopToChr was passed an object which was not a Character.
2165	#hostErrNoPlusInfinity	GciOopToFlt detected a value of positive infinity, but the host floating point does not represent this IEEE value.
2166	#hostErrNoMinusInfinity	GciOopToFlt detected a value of negative infinity, but the host floating point does not represent this IEEE value.
2167	#hostErrNoPlusQuietNan	GciOopToFlt detected the Float PlusQuietNaN, but the host floating point does not represent this IEEE value.
2168	#hostErrNoMinusQuietNan	GciOopToFlt detected the Float MinusQuietNaN, but the host floating point does not represent this IEEE value.

2169	#hostErrNoPlusSignalingNan	GciOopToFlt detected the Float PlusSignalingNaN, but the host floating point does not represent this optional IEEE value.
2170	#hostErrNoMinusSignalingNan	GciOopToFlt detected the Float MinusSignalingNaN, butthe host floating point does not represent this optional IEEE value.
2171	#rtErrUalibLoadFailed	An attempt to load a user action library failed because: <arg1>.
2172	#hostErrMagnitudeOutOfRange	GciOopToFlt detected a value which has a magnitude greater than a C double precision variable.
2173	#authErrSegWriteSeg	An attempt was made to write the segment <arg1> with insufficient authorization.
2174	#authErrSegReadSeg	An attempt was made to read from the segment <arg1> with insufficient authorization.
2175	#assocErrPathTooLong	The path <arg1> has more than 1024 characters.
2176	#repErrFileNameTooBig	The filename <arg1> has more than <arg2> characters.
2177	#rtErrSemaphore	A semaphore operation failed on semaphore with index <arg1>. Reason: <arg2>.
2178	#rtErrPasswordTooBig	A password was specified with more than 255 characters.
2179	#errNotSameClassHist	Migration is not allowed because the classes <arg1> and <arg2> do not have identical class histories.
2180	#classErrMethDictLimit	The class <arg1> has more than 1500 methods in its method dictionary.
2181	#rtErrShrPcDetach	Error occurred during SharedPageCache detach: <arg1>

2182	#repErrCantDispose	Unable to dispose of the file <arg1>.
2183	#rtErrInternal	Please report to your GemStone Administrator. System Runtime error number: <arg1> with arguments: <arg2> <arg3> <arg4> <arg5> <arg6> <arg7> <arg8> <arg9> <arg10>.
2184	#rtErrBadStreamColl	An attempt was made to create an instance of <arg1> from the stream <arg2>, but the collection in the stream is not a <arg3>.
2185	#rtErrBadFormat	An attempt was made to create an instance of <arg1> from <arg2> but the format is incorrect.
2186	#rtErrShrpcCompatibility	The compatibility level of the SharedPageCache Monitor<arg1>does not match that of the executable trying to attach<arg2>
2187	#segErrBadGroup	<arg1> is not currently a group.
2188	#rtErrBadPriv	<arg1> is not a valid privilege.
2189	#rtErrResolveSymFailed	GciResolveSymbol failed; the symbol is not defined
2190	#rtErrSymAlreadyDefined	<arg2> is already defined.
2191	#rtErrSymNotFound	<arg2> could not be found in the symbol list for <arg1>.
2192	#rtErrEofOnReadStream	End of stream was encountered in ReadStream: <arg1>.
2193	#assocErrSortOddLengthArray	An illegal sorting was specified: <arg1>.
2194	#assocErrBadDirection	A sort direction must be 'ascending' or 'descending', not <arg1>.
2195	#rtErrConfigReadOnly	Smalltalk DB access to the configuration parameter <arg1> is read-only.
2196	#rtErrBadFormatSpec	<arg2> is an illegal formatting array for <arg1>.
2197	#hostErrFileExport	GemStone cannot export the string <arg1> to the file <arg2> in the server OS.

2198	#hostErrFileImport	GemStone cannot import the file <arg1> from the server OS.
2199	#hostErrFileDirectory	GemStone cannot fetch the server OS directory <arg1>.
2200	#hostErrFileDelete	GemStone cannot delete the file <arg1> from the server OS.
2201	#hostErrPerform	GemStone cannot execute the string <arg1> on the server OS shell.
2202	#rtErrSigMsgTooBig	Attempt to send a string of size <arg2> bytes as argument to a signal. The maximum allowed size is <arg3> bytes.
2203	#gciErrOpInProgress	A GemBuilder for C (GCI)operation was requested before the current nonblocking call was completed.
2204	#gciErrNoStartCmd	A request was made to complete a non-blocking GemBuilder for C call when no such call was initiated.
2205	#objErrBadClusterBucket	The clusterId <arg1> is invalid. Possible bad ClusterBucket is <arg2>. Max legal clusterId is <arg3> .
2206	#rtErrEpochGcArraySize	The number of elements in the array used to set the Epoch Garbage Collector information is incorrect. The size of <arg1> should be 4.
2207	#objErrResultNscTooBig	The Nsc operation failed; the size of the result would have exceeded the maximum size for an Nsc.
2208	#hostErrMemoryAlloc	Host memory allocation failed; there is insufficient primary memory and/or swap space.
2209	#gciErrCatchBuffNotFound	A non-existent catch buffer was specified.
2210	#gciErrCatchBuffLimit	The catch buffer level must be in the range 0 to 20. An invalid level was specified.
2211	#objErrNotBoolean	GciOopToBool was passed an object that was not either true or false.

2212	#rtErrUncompiledMethod	The method <arg3> with selector <arg1> in class <arg2> is obsolete after schema modification or repository conversion and must be recompiled.
2213	#rtErrMustBeSystemUser	An operation was attempted that may only be performed by SystemUser.
2214	#rtErrBadPassword	The given password is not the password of <arg1>.
2215	#gciErrTravObjNotFound	The given object was not found within the given traversal buffer.
2216	#gciErrTravCompleted	A continuation of a traversal was attempted when there was no traversal in progress.
2217	#gciErrTravBuffTooSmall	The given traversal buffer length must be of sufficient size to report at least one object.
2218	#rtErrPathToStrIvname	Path to String conversion starting with class <arg1> failed at term <arg2> because there is no named instVar at offset <arg3> in class <arg4> which has <arg5> named instVars.
2219	#objErrNegativeCount	In GciFetchBytes, GciFetchIdxOops, or GciFetchNamedOops, a negative count of <arg1> was specified.
2220	#gciErrResultPathTooLarge	GciStrToPath or GciPathToStr, result of size <arg1> is larger than specified max size of <arg2>.
2221	#gciErrFetchPathFail	GciFetchPaths failed on a path; OOP_ILLEGAL substituted in result array.
2222	#rtErrStrToPathIvname	String to path conversion starting with class <arg1> failed on path term <arg2> because there is no instVar named <arg3> in class <arg4>.
2223	#rtErrStrToPathConstraint	Path conversion starting with class <arg1> failed at path term <arg2> due to lack of constraints.
2224	#gciErrBreakCanceledMsg	The command was ignored because of a hard break.

2226	#lgcErrSync2	The second byte of the two-byte synchronization sequence was wrong in the network.
2227	#lgcErrBytePacketTooLong	A byte array packet was received over the network that was too long.
2228	#lgcErrArgSizeInconsistent	An array argument was received on the network that had a length different from that specified with the argument.
2229	#lgcErrOopPacketTooLong	An oop array packet was received over the network that was too long.
2230	#lgcErrPacketKindBad	A packet of the wrong kind was received. Expected: <arg1>, Received: <arg2>.
2231	#lgcErrExpectedContinue	A packet kind of <arg1> was received rather than the continue packet that was expected.
2232	#lgcErrExpectedEnd	A packet kind of <arg1> was received rather than the end packet that was expected.
2233	#lgcErrPacketKindUnknown	An unknown packet kind was received: <arg1>.
2234	#lgcErrExpectedCmd	A packet kind of <arg1> was received rather than the command packet that was expected.
2235	#hostErrLogFileNotOpened	The log file was not opened before adding messages to it.
2236	#classErrMaxInstVars	An attempt was made to create a subclass of <arg1> with more than 255 instance variables specified in <arg2>.
2237	#rtErrTooManyErrArgs	System | signal:args:signalDictionary: was sent with an Array argument containing <arg2> arguments, but it is limited to 10.
2238	#objErrBadSize	For class <arg1> the size <arg2> is illegal.
2239	#lgcErrInconsistentSize	Inconsistent size information received. Specified size = <arg1>, implied size = <arg2>.
2240	#lgcErrInconsistentObjKind	Inconsistent implementation received. Class = <arg1>, implementation = <arg2>.

2241	#rtErrStartGc	A process is already running as the Garbage Collector or a session attempting to run the Reclaim or EpochGc is not the Gc process.
2242	#rtErrBadArgKind2	The object <arg1> was neither of class <arg2> nor <arg3>
2243	#lgcErrSequenceMismatch	Sequence number mismatch for an IPC response packet.
2244	#rtErrExceptBlockNumArgs	The block <arg1> which takes <arg3> arguments, cannot be used as an Exception block. Exception blocks require <arg2> arguments.
2245	#rtErrGciStoreFloat	An illegal GemBuilder for C (GCI)store into the float object <arg1> was detected. You must use GciStoreBytesInst with the correct class argument.
2246	#rtErrGciStoreClassMismatch	Mismatch between class in store traversal buffer and class in Repository for object <arg1>. Class in buffer is <arg2> , class in Repository is <arg3>
2247	#rtErrNscParticipatesInMultipleTerms	The NSC <arg1> is not allowed to participate in more than one term in an index path (<arg2>).
2248	#rtErrCommitDbInRestore	Commits are not allowed while a restore from backups or transaction logs is in progress.
2249	#rtErrCommitDisallowed	A previous error occurred during object manager recursion to GemStone Smalltalk (possibly during an indexing maintenance operation). This transaction must be aborted.
2250	#tranLogIoError	I/O error when writing the transaction log. This transaction must be aborted.
2251	#lgcErrPacketKind	Invalid IPC packet kind.

2252	#rtErrLoadSaveStack	Error encountered while saving or reloading Smalltalk DB execution state. Execution cannot continue.
2253	#rtErrUnknownBytecode	Unknown opcode = <arg1> encountered during Smalltalk DB execution.
2254	#errSesBlockedOnOutput	Attempt to send signal to session <arg1> failed. Buffer from stone to that session is full.
2255	#errPrimNotSupported	Primitive failure, primitive for receiver= <arg1> selector= <arg2> is not supported in this executable. You must run "<arg3>" in order to use this method.
2256	#authErrSegCurrentSeg	No authorization write in your current segment <arg1>.Resetting current segment to default login segment.
2257	#authErrSegSetCurrentSeg	No authorization to set the current segment to <arg1>.
2258	#rtErrPrimFailed	Primitive failed , selector <arg2> receiver <arg1> .
2259	#gciErrExecClientUserAct	Request to invoke client userAction named <arg1>, invocation failed.
2260	#gciErrActiveSessions	You may not install a user action after logging in. Action name: <arg1>
2261	#objErrCorruptObj	The object with object id <arg1> is corrupt. Reason: <arg2>
2262	#gciErrMaxActionArgs	You are requesting <arg2> arguments to a user action when only <arg3> are allowed. Name: <arg1>.
2263	#gciErrBadNumActionArgs	The method or user action <arg1> takes <arg2> arguments, not <arg3> as passed.
2264	#gciErrUserActionPending	You attempted an illegal GemBuilder for C (GCI)operation during a user defined action <arg1>.

2265	#gciErrBadNumLclActionArgs	You invoked a local user action with an incorrect number of arguments. Name, correct, and actual: <arg1><arg2><arg3>.
2266	#rtErrInstvarAddToNsc	An attempt was made to add a named instance variable to the class <arg1> which has a format of NSC.
2267	#rtErrVaryingConstrBytes	An attempt was made to constrain the indexable portion of the class <arg1> which has a format of Bytes.
2268	#rtErrVaryingConstrNonidx	An attempt was made to constrain the indexable portion of the class <arg1> which is not indexable.
2269	#rtErrInstvarAddToBytes	An attempt was made to add a named instance variable to the class <arg1> which has a format of Bytes.
2270	#rtErrClassNotModifiable	The class <arg1> is not modifiable.
2271	#rtErrAddDupInstvar	The name of the new instance variable, <arg2>, would duplicate the name of an existing instance variable in class <arg1>.
2272	#rtErrNotASubclassOf	The class <arg1> is neither identical to nor a subclass of the class <arg2>.
2273	#rtErrConstrNotSubclassOf	In a class modification operation, the new constraint <arg1> was neither identical to, nor a subclass of, the inherited constraint <arg2>.
2274	#rtErrConstrNotAClass	The new constraint <arg1> was not a class.
2275	#rtErrObjInvariant	The object <arg1> is invariant.
2276	#classErrDupVarConstr	Two constraints, <arg1> and <arg2>, were specified upon the indexable portion of the class. Only one constraint is allowed.
2277	#bkupErrLoginsEnabled	A restore operation requires that logins be disabled. Run System suspendLogins before doing restores.

2278	#classErrRemoveInherIv	In class <arg1>, an attempt was made to remove the instance variable <arg2> which is inherited from a superclass.
2279	#concurErrInvalidMode	<arg1> is not recognized as a valid Concurrency Control Mode.
2280	#classErrSelectorLookup	The message <arg4> sent to object <arg1> was found in class <arg2>. It should have been found in class <arg3>
2281	#rtErrBadEUCFormat	The bytes of the EUC string with biased id <arg1> are not in EUC format.
2282	#rtErrGaijiNotSupported	A Gaiji character was encountered in the JapaneseString <arg1> but Gaiji is not supported.
2283	#rtErrInvalidArgClass	<arg1> is not one of the class kinds in <arg2>.
2284	#rtErrSizeLimit	The object <arg1> was referenced with a byte index <arg2> which is out of range.
2285	#rtErrNoEUCRep	<arg1> cannot be represented in EUC format.
2286	#rtErrBadEUCValue	<arg1> is not a valid EUC value and does not correspond to a JISCharacter.
2287	#rtErrInvalidLang	The compiler language environment <arg1> is invalid.
2288	#rtErrInvalidIndexPathExpression	The following string is an invalid term in a path expression: <arg2>
2289	#rtErrDependencyListTooLarge	A dependency list <arg1> is too large.
2290	#rtErrMaxCommitRetry	There were too many attempts to commit after a concurrency conflict or after failing to resolve RC conflicts. You must abort before attempting to commit again.
2291	#rtErrInvalidArgument	The object <arg1> was an invalid argument to a method, reason: <arg2>

2292	#rtErrPathNotTraversable	The class <arg2> does not have an inst var <arg3> on the index path.
2293	#rtErrBtreeReadStreamEndOfStream	An attempt was made to read beyond the end of the stream.
2294	#rtErrObjectPathTermNotInDependencyList	The object <arg1> did not have the path term <arg2> in its dependency list.
2295	#rtErrObjectInvalidOffset	The object <arg1> does not have an instance variable with the given name <arg2>
2296	#rtErrObjectNoDependencyList	The object <arg1> does not have a dependency list.
2297	#rtErrIndexDictionaryEntryNotInDictionary	An entry for the key/term/value (<arg2>/<arg3>/<arg4>) was not present in the dictionary <arg1>
2298	#rtErrPathTermObjectNotAnNsc	The object <arg1> traversed along an index path through a set-valued inst var was not an Nsc.
2299	#rtErrIdentityIndexCannotInvokeRangeOperation	An attempt was made to use a range operation (<, >, =, <=, or >=) on a path expression only supported by an identity index.
2300	#rtErrRangeEqualityIndexInvalidClassKindForBtree	Attempt to insert a key into the btree that was an invalid class for which the btree was created.

2301	#rtErrRangeEqualityIndexObjectNotInBtree	
		An entry for the key/value pair (<arg2>/<arg3>) was not present in the index.
2302	#errNoBackupInProgress	A backup or restore continuation was attempted before executing either restoreFrom: or fullBackupTo:MBytes: .
2303	#bkupErrOpenFailed	An attempt to open file <arg1> for <arg2> failed because <arg3>.
2304	#bkupErrMbyteLimitBadRange	
		The byte limit specified <arg1> is out of the allowable range from <arg2> to <arg3> Mbytes.
2305	#bkupErrWriteFailed	An attempt to write to file <arg1> failed because <arg2>.
2306	#bkupErrInProgress	An attempt was made to start a full backup, but a backup is currently in progress by another session.
2307	#bkupErrReadFailed	Read error during restore, <arg2>, in record <arg3> of file <arg1>
2308	#rtErrBagNoConstraintAlongPath	
		An attempt was made to create an index along the path <arg2> with no constraint.
2309	#rtErrBagClassDoesNotSupportRangeOperators	
		An attempt was made to create an equality index with a class <arg2> that does not support range operators.
2310	#rtErrBagOnlySelectBlockAllowed	
		Only select blocks are allowed for selectAsStream.
2311	#rtErrBagOnlyOnePredicateAllowed	
		Only one predicate is allowed for selectAsStream.

2312	#rtErrBagNoRangeIndexOnPathExpression	The path expression in the predicate for selectAsStream does not have a range equality index.
2313	#rtErrBagInvalidPredicateForStreamSelection	The predicate for selectAsStream was invalid.
2314	#rtErrBagOperationNotSupportedForStreamSelection	The comparison operation in the predicate for selectAsStream is not supported.
2315	#rtErrBagInvalidSortSpecification	Unable to sort using the sort specification: <arg2>
2316	#rtErrIndexAuthErrSegRead	An attempt was made to read the object using index <arg2> in segment <arg3> with insufficient authorization.
2317	#objErrTime_tOutOfRange	Given time_t is out of range; must be greater than or equal to zero.
2318	#genericError	User defined error, <arg2>
2319	#rtErrMethodProtected	Illegal attempt to execute a protected method.
2320	#rtErrBadConstraintForMigration	The object <arg1> cannot be migrated because inst var at offset <arg2> participates in an index with a constraint of <arg3>.
2321	#rtErrPreventingCommit	<arg1> This error occurred during index maintenance. Consequently, this transaction must be aborted.
2322	#rtErrCantBecomeSelfOnStack	The object <arg1> is present on the Smalltalk DB stack as "self", and cannot participate in a become.

2323	#rtErrObjectProtected	Illegal attempt to fetch/store into a protected object.
2324	#rtErrNewTranlogDirFail	Attempt to define new transaction log directory failed, reason: <arg1>
2325	#errCommitWhileBackupInProgress	A commit or abort was attempted while a multi-file full backup is in progress. To cancel the backup use the abortFullBackup method. To continue the backup use the continueFullBackupTo:MBytes: method.
2326	#errUnconvertedObject	Incomplete conversion from previous version of GemStone, object: <arg1> reason: <arg2>
2327	#rtErrLastConstraintNotBoolean	The given SelectBlock must contain a Boolean expression.
2328	#rtErrCommitProhibitingError	<arg1> This error occurred during object manager recursion to GemStone Smalltalk. This transaction must be aborted.
2329	#rtErrAttemptToPassivateInvalidObject	Attempted to write <arg1> to a passive object. nil was written instead.
2330	#rtErrTimeToRestoreToArg	timeToRestoreTo: failed, reason: <arg1>
2331	#lockErrAllSymbols	Users are not allowed to lock AllSymbols
2332	#gciErrSymbolFwdRef	User attempted to create a forward reference, oop = <arg1> to a Symbol or fill in the state of an existing forward reference as a Symbol
2333	#rtErrChangeSymbol	Changing the class of an object <arg1> from or to class Symbol is disallowed
2334	#rtErrObjVariant	The object <arg1> is not invariant.
2335	#rtErrAlreadyHasSubclasses	In disallowSubclasses, <arg1> already has subclasses.

2336	#clientForwarderSend	Message to forward to client, rcvr: <arg1> selector: <arg2> args: <arg3>
2337	#rtErrBadSize	Invalid object size, required size <arg2> actual size <arg3> for object <arg1>
2338	#rtErrFreeSpaceThreshold	The Repository is currently running below the freeSpaceThreshold.
2339	#rtErrTranlogDirFull	The tranlog directories are full and the stone process is waiting for an operator to make more space available by either cleaning up the existing files (copying them archive media and deleting them) or by adding a new tranlog directory.
2340	#objErrDictConstraintViolation	Attempt to store <arg2> of class <arg4> into <arg1> constrained to hold only instances which are a kind of <arg3>.
2341	#rtMaxPasswordSize	The maximum size of a password is <arg2> characters. The string <arg1> is too large.
2342	#rtMinPasswordSize	The minimum size of a password is <arg2> characters. The string <arg1> is too small.
2343	#rtMaxConsecutiveChars	A password may not have more than <arg2> consecutive characters. The substring <arg3> is invalid in the password <arg1>
2344	#rtMaxRepeatingChars	A password may not have more than <arg2> repetitions of a character. The substring <arg3> is invalid in the password <arg1>
2345	#rtMaxCharsOfSameType	A password may not have more than <arg2> consecutive <arg3> characters. The substring <arg4> is invalid in the password <arg1>
2346	#rtDisallowedPassword	A password may not have the value <arg1>
2347	#rtPasswordExpireWarning	The password of the current UserProfile is about to expire.

2348	#errLogDirNotExist	No directory or raw device exists with the name <arg1>
2349	#errArgTooSmall	The object <arg1> is too small. The minimum size is <arg2> .
2350	#errNoStructuralUpdate	GemBuilder for C update operation <arg2> is not supported on object <arg1>
2351	#rtObsoleteClass	New instances of this obsolete class are not allowed.
2352	#rtErrLocalSessionFailedCommit	Local session failed to commit after remote sessions voted to commit.
2353	#rtErrRemoteSessionFailedCommit	Remote session <arg2> failed to commit after voting affirmative.
2354	#rtErrNoElemDetected	The object <arg2> was not detected in <arg1>.
2355	#rtErrDateTimeOutOfRange	The resulting DateTime object would be out of range.
2356	#rtErrObjNotFound	The object <arg2> was not found in <arg1>.
2357	#rtErrFailedStnCfgWrite	The operation should update the stone configuration file, but the write failed. See stone log for more details.
2358	#gciErrNoUserAction	Attempt to call a user action that is not registered with this virtual machine, user action name: <arg1>
3001	#rtErrAbortTrans	The transaction was aborted by the user.
3006	#abortErrGarbageCollection	Garbage collection aborted, reason: <arg1>, conflict code: <arg2>. Garbage collection was aborted and should be tried again later.
3007	#abortErrUndefPomObj	A reference was made to object <arg1> in the permanent object manager even though the object has not been defined.

3008	#bkupErrRestoreSuccessful	Restore from full backup completed with <arg1> objects restored and <arg2> corrupt objects not restored.<arg3>
3009	#bkupErrBackupNotComplete	The backup was successful, but was not completed due to a byte limit restriction; a partial backup file was created containing <arg1> objects and is <arg2> bytes in size.
3011	#bkupErrRestoreLogSuccess	Restore from transaction log succeeded.
3012	#bkupErrRestoreLogFail	Restore from transaction log failed. Reason: <arg1>
3016	#bkupErrRestoreCommitFailed	Commit failed, restore is unusable, <arg1>
3020	#abortErrFinishedMark	Successful completion of markForCollection. <arg1> live objects found. <arg2> possible dead objects, occupying <arg3> bytes, may be reclaimed.
3021	#abortErrFinishedObjAudit	Completed execution of object audit. <arg1> objects contained errors.
3031	#abortErrLostOtRoot	This error indicates that when running outside of a transaction stone signaled the gem to indicate that it did not respond in time so it revoked access to the object table root.
4001	#gsErrBadRootPage	The root page of the Repository is bad. A disk access of the root page encountered a disk read error, or found corrupted data.
4002	#repErrReposFull	The logical Repository is full; the Repository is <arg1>.
4003	#repErrNoFreePages	No free pages were found after growing the Repository <arg1>.
4004	#hostFatalErrOutOfMem	Host memory allocation failed during <arg1>. Insufficient primary memory and/or swap space.

4005	#gsErrCorruptObjSize	The object <arg1> is corrupted and has size <arg2>. The correct size is <arg3>.
4006	#repErrBadDbfSwizzle	An invalid swizzle transform was detected in an extent long transform: <arg1> short transform: <arg2>
4007	#gciErrActionDefined	You have already installed the user action, <arg1>.
4008	#errUserProfileLost	The UserProfile with object id <arg1> has been garbage collected as a result of a successful restore or by a commit of another session
4009	#gsErrShrpcConnectionFailure	Detected a connection failure from the SharedPageCache monitor: Error text: <arg1>
4010	#gsErrShrpcUnexpectedNetEvent	Detected an unexpected <arg1> event from the SharedPageCache monitor.
4011	#gsErrShrpcInvalidConfig	The process tried to login to a Stone with the config file specifying that the SharedPageCache should NOT be used when the Stone already has a SharedPageCache active on the host machine
4032	#errTranLogOpenFail	Unable to open next transaction log file for writing.
4034	#gsErrStnNetProtocol	A protocol error occurred on the Gem-Stone network, failure code = <arg1> .
4035	#gsErrStnNetLost	An end of file was received over the Gem-Stone network, indicating that the network is being shut down.
4038	#gsErrDisconnectInLogin	Stone disconnected during an attempt to login, probably because logins are disabled or max users are already logged in.
4039	#gsErrMaxSessionsLimit	Login failed because the system was full.

4040	#lgcErrIncompatGci	The version of GemBuilder for C (GCI)is not compatible with the version of the Gem
4042	#hostErrCantSpawn	HostFork() attempted to spawn a process but was unable to do so.
4050	#gsActiveUserLimitExceeded	
		Too many sessions already logged in with this userId.
4051	#gsErrLoginDenial	Login failed. The userId/password combination is invalid or expired.
4053	#gsErrLoginsDisabled	The login failed because all logins are currently disabled.
4057	#gsErrStnShutdown	Stone is shutting down.
4059	#gsErrSessionShutdown	The Data Curator is shutting down this session.
4060	#gsErrGemNormalShutdown	Gem is shutting down normally.
4061	#gsErrGemFatalShutdown	Gem did not shut down normally.
4062	#gsErrNoMoreOops	GemStone system ran out of oops.
4065	#netErrNoSuchStn	The given GemStone does not exist.
4100	#gciErrBadSessionId	GemBuilder for C (GCI) was called with an invalid SessionId.
4101	#gciErrUnexpectedLnkErr	Fatal unexpected error in Linkable GemBuilder for C (GCI)session while GemBuilder for C (GCI) call in progress on another RPC session
4102	#gciErrInternalRpc	A logic error was detected in the implementation of an RPC GemBuilder for C call. Please report to your GemStone Administrator.
4126	#fatalErrInternal	Please report to your GemStone Administrator. System Fatal error number: <arg1> with arguments: <arg2> <arg3> <arg4> <arg5> <arg6> <arg7> <arg8> <arg9> <arg10>.

4136	#netErrConnectionRefused	The connection was refused - check the Stone name.
4137	#netErr	An undefined network error. System error text is appended to message:
4138	#repErrSystemRepNotAttached	The system Repository is not attached.
4140	#authErrSegLoginSeg	Commit succeeded, but there is no authorization to write in default login segment <arg1>.
6001	#rtErrPause	Execution has been suspended by a "pause" message.
6002	#rtErrStep	Single-step breakpoint encountered.
6003	#rtErrSoftBreak	A soft break was received.
6004	#rtErrHardBreak	A hard break was received.
6005	#rtErrCodeBreakpoint	Method breakpoint encountered.
6006	#rtErrStackBreakpoint	Stack breakpoint encountered on return from method or block.
6007	#rtErrCommitAbortPending	A transaction commit or abort is pending.
6008	#rtErrSignalCommit	This error indicates that a member of the notifySet has been committed and was added to the signaledObjects set.
6009	#rtErrSignalAbort	This error indicates that when running outside of a transaction stone signaled the gem to request an abort
6010	#rtErrSignalGemStoneSession	The signal <arg2> was received from sessionSerialNumber = <arg1> and the message string associated with the signal is <arg3>
6011	#rtErrUncontinuable	Execution cannot be continued. An attempt was made to continue past an uncontinuable error, or recursive Exception blocks may have left the Smalltalk DB stack in an unusable state.

UNUSED ERROR NUMBERS:

1 to: 1000 ; 1064 to: 2000 ; 2360 to: 3000 ;

3002 to: 3005 ; 3010 to: 3010 ; 3013 to: 3015 ;

3017 to: 3019 ; 3022 to: 3030 ; 3032 to: 4000 ;

4012 to: 4031 ; 4033 to: 4033 ; 4041 to: 4041 ;

4043 to: 4043 ; 4046 to: 4049 ; 4052 to: 4052 ;

4054 to: 4056 ; 4058 to: 4058 ; 4063 to: 4064 ;

4066 to: 4099 ; 4103 to: 4125 ; 4127 to: 4135 ;

4139 to: 4139 ; 4141 to: 6000

Index

Symbols

* (in a path) 5-31
+ (GsFile) 9-5
+ (String) 4-27
^ 11-18

A

abortErrFinishedObjAudit 11-25
aborting
  errors B-1
  receiving a signal from Stone 6-12
  releasing locks when 6-23
  transaction 6-11
  views and 6-11
AbortingError 3-5
`abortTransaction` (System) 6-11
abstract superclass
  SequenceableCollection 4-13–4-19
accessing
  method 13-2
  operating system from Smalltalk 9-1
  pool dictionaries 13-13
  SequenceableCollections with streams 4-42
  variables 13-3, 13-13
acquiring locks 6-16
`add:` (RcBag) 6-29
`add:` (RcQueue) 6-29
`add:` (String) 4-28
`add:withOccurrences:` (IdentityBag) 4-31
`addAll` (String) 4-28
`addAll:` (GsFile) 9-5
`addAllToNotifySet:` (System) 10-7
`addCategory:` (Behavior) 13-10

adding
  category 13-10
  method 13-2
  to a SequenceableCollection 4-14, 4-15
  to notify set 10-5–10-8
  to symbol lists 3-4
  users to symbol lists 3-12
addNewVersion: (Object) 8-6
addObjectToBtreesWithValues: (Object) 5-15
addPrivilege (UserProfile) 7-31
addPrivilege: (UserProfile) 7-31
addPrivileges: (UserProfile) 7-32
addToCommitOrAbortReleaseLocks-Set: (System) 6-23
addToCommitReleaseLocksSet:(System) 6-23
addToNotifySet: (System) 10-5
allClassVarNames (Behavior) 13-13
AllClusterBuckets 12-5, 12-7
allInstances (ClusterBucket) 12-6
allInstances (Repository) 8-9
allInstVarNames (Behavior) 13-13
allSelectors (Behavior) 13-7
allSharedPools (Behavior) 13-13
AND (in selection blocks) 5-9
application objects 6-12, 6-13
archiving data objects 9-10
arguments A-12
  block A-20
arithmetic
  mixed-mode 12-21
Array 4-14–4-23
  comparing with C or Pascal 4-20
  constructors A-15
  creating 4-20
  invariance 4-22
  large, and efficiency 4-22
  literal 4-22, A-7
  performance of 12-21
assigning
  class history 8-6
  cluster buckets 12-11
  migration destination 8-7
assignment A-9
Association 4-10
associative access 5-1–5-34
  comparing strings 5-10
  nil values 5-30
asterisk
  as wild card character 9-8
  in a path 5-31
at:equals: (String) 4-25
atEnd (RangeIndexReadStream) 5-17
atomic objects 10-7
  clustering and 12-10
  disk page of 12-14
  indexing and 5-24
  locking and 6-16
authorization 7-29
  assigned to segment 7-9
  clustering 12-11
  error while redefining class 8-5
  indexing and 5-28
  locking and 6-16
  migration and 8-13
  owner 7-15
  read/write 7-10
  segments and 7-14
  world 7-11
authorizations, instance variable for Segment 7-12
automatic transaction mode 6-7
  defined 6-7

B

Bag
  as relation 5-2
  converting to RcBag 5-16
  maximum size 4-30
  methods implemented, compared to

RcBag 6-29
balanced tree
conflict on 6-13
defined 5-3
beginTransaction (System) 6-8
Behavior 13-1–13-18
binary messages A-12, A-13
binding
source code symbol 3-2–3-12
bkupErrRestoreSuccessful 11-25
blocks A-19
arguments A-20
complexity of 12-22
conditional execution A-22
empty A-20
executing A-19
literal A-19
optimized 12-22
repeated execution A-23
selection 5-4–5-15
using curly braces 5-4
BNF syntax for Smalltalk A-27
Boolean 10-7
locking and 6-16
operators in queries 5-9
segment of 7-14
branching A-22
breadth-first clustering example 12-10
bucketWithId: (ClusterBucket) 12-7

C

C programming language
comparing arrays 4-20
cache 12-24–12-27
changing size of 12-25–12-27
estimating available size 12-23
temporary object space 12-24
cancelMigration (Object) 8-8
canUnderstand: (Behavior) 13-7
caret 11-18
cascaded messages A-14
case of variable names A-7
case-sensitivity A-2
category
adding 13-10
for errors 11-9
moving methods among 13-10
removing 13-10
category: number: do: (Exception) 11-15
categoryNames (Behavior) 13-10
changed object notification 10-3
changes, receiving notification of 10-3, 10-10–10-11
by polling 10-12
changeToSegment: (Object) 7-14
changing
cache sizes 12-24–12-27
cluster bucket 12-6
database
notification of 10-5–10-11
notification of<$startange> 10-9
transaction modes and 6-7
invariant objects 8-2
memory allocated to caches 12-25
objects
notification of 10-3–10-11
visibility of to other users 6-8
privileges 7-31
segment before committing transaction 7-6
Character 10-7
literal A-5
locking and 6-16
segment of 7-14
Class 13-1–13-18

class
  clustering 12-12
  determining structure 13-4
  history 8-3–8-7
    constraints and 8-6
  migrating 8-10
  names
    versioning and 8-6
  predefined 2-12
  RcHashDictionary, indexing and 6-27
  redefining 8-2–8-26
  reduced conflict 6-26, 12-25
    collections returned by selection 5-16
    when to use 6-27
  renaming 8-3
  storage and reducing conflict 6-26
  versions 8-2–8-3
class version, defined 8-3
ClassHistory 8-3–8-7
  assigning 8-6
  constraints on 8-6
  determining 8-5
`classVarNames` (Behavior) 13-13
`cleanupMySession` (RcQueue) 6-30
`clearCommitOrAbortReleaseLocksSet` (System) 6-24
`clearCommitReleaseLocksSet` (System) 6-24
clearing notify set 10-8
`clearNotifySet` (System) 10-8
client, defined 9-2
`cluster` (Object) 12-8
`clusterBehavior` (Behavior) 12-13
`clusterBehaviorExceptMethods:` (Behavior) 12-13
ClusterBucket 12-3–12-15
  assigning 12-11
  changing 12-6
  concurrency and 12-7–12-8
  default 12-6
  describing 12-7
  determining current 12-6
  extent of 12-4
  indexing and 12-8
  using several 12-11
`clusterBucket` (Object) 12-11
`clusterBucket:` (System) 12-6
`clusterDepthFirst` (Object) 12-11
`clusterDescription` (Behavior) 12-13
`clusterId` (ClusterBucket) 12-6
`clusterInBucket:` (Object) 12-11
clustering 12-2–12-15
  as factor in performance 12-2
  atomic objects and 12-10
  authorization for 12-11
  breadth-first, example 12-10
  buckets for 12-3
    extents of 12-4
  classes 12-12
  concurrency conflict and 12-7
  depth-first 12-11
  global variables 12-5
  instance variables 12-9
  kernel class methods 12-5
  maintaining 12-15
  messages (table) 12-12
  recursion and 12-10
  shared page cache size and 12-27
  source code for kernel classes 12-5
code formatting A-25

Collection
as variable in a path 5-31–5-33
common protocol 4-4
constraining elements 4-8
creating efficiently 4-5
enumerating 4-6
errors while locking 6-20
hierarchy 4-3
indexing and clustering 12-8
indexing and write set 6-10
locking efficiently 6-19
rejecting elements 4-7
returned by selection blocks 5-16
searching 4-7
efficiently 5-1–5-34
selecting elements 4-7
sorting 4-33
streaming over 4-43
subclasses 4-10–4-42
unordered 4-30–4-37
updating indexed 5-27
combining expressions A-13
commands, executing operating system 9-9
comment A-3
`commitAndReleaseLocks` (System) 6-23, 6-24
`commitOrAbortReleaseLocksSet-Includes:` (System) 6-26
`commitReleaseLocksSetIncludes:` (System) 6-26
committing a transaction 6-2
after changing segments 7-6
after recomputing index segments 5-28
effects of 6-8
failure 6-10, 6-11
frequency of, to optimize performance 12-23
moving objects to disk and 12-11
performance 6-27
releasing locks when 6-23
when 6-2
write locks to guarantee success 6-15
communicating between sessions 10-2–10-23
communicating from session to session
diagram 10-16
comparing
instances of kernel classes 5-7
InvariantStrings 4-29
literal strings 4-29
messages and selection block predicates 5-10
nil 5-8
selection blocks 5-7
SequenceableCollection 4-17
Strings 4-29
`compileAccessingMethodsFor:` (Behavior) 13-2
`compiledMethodAt:` (Behavior) 13-7
CompileError 3-5
`compileMethod: dictionaries: category:` (Behavior) 13-5
compiler error numbers 13-6
compiling methods programmatically 13-5
concatenating strings 4-27, 12-22
concurrency 6-1
cluster buckets and 12-7–12-8
conflict 6-7
concurrency control
optimistic 6-6–6-11
when to use 6-5
pessimistic 6-13–6-14
CONCURRENCY_MODE 6-5
conditional
execution and blocks A-22
repetition A-23
selection A-22
configuration parameter 12-24–12-25
CONCURRENCY_MODE 6-5
GEM_PRIVATE_PAGE_CACHE_KB 12-25
GEM_TEMPOBJ_CACHE_SIZE 12-25
SHR_PAGE_CACHE_SIZE_KB 12-25
STN_GEM_ABORT_TIMEOUT 6-12
STN_PRIVATE_PAGE_CACHE_KB

12-25
conflict 6-7–6-26
checking 6-5
keys (table) 6-9
on indexing structure 6-11
read set 6-4
read/write 6-5, 6-27
reducing 6-26–6-31, 12-25
performance 6-27
semantics of 6-26
while overriding a method 6-10
with cluster buckets 12-7
write set 6-4
write/write 6-5, 6-27
conjoining predicate terms 5-9
consistency of database, preserving 6-3
constants A-8
constrained collections, sorting 4-33
constraining
class history 8-6
elements of a collection 4-8
instance variables 2-5
paths 5-6
constraints 2-5
and instance migration 8-20
circular 2-7
determining 4-9
error with classes having the same name 8-6
full 5-6
inheritance of 2-6
partial 5-6
constructors, array A-15
`contentsAndTypesOfDirectory: onClient` (GsFile) 9-8
`contentsOfDirectory: onClient:` (GsFile) 9-8
context exception, defined 11-6
`continueTransaction` (System) 6-12
control, flow of 4-6
copying
objects 12-22
`cr` (GsFile) 9-5
`createDictionary:` (UserProfile) 3-6
`createEqualityIndexOn:` (Bag) 5-22
`createEqualityIndexOn:` (Collection) 5-24, 5-29
`createIdentityIndexOn:` (Bag) 5-21
creating
arrays 4-20
equality indexes 5-22
files 9-3
identity indexes 5-21
subclass A-2
symbol list dictionaries 3-6
curly braces for selection blocks 5-4
`currentClusterBucket` (System) 12-6
`currentSegment:` (System) 7-6
`currentSessionNames` (System) 10-17
`currentSessions` (System) 10-19
customizing data retention during migration 8-16

D

data
efficient retrieval 12-2–12-15
retaining during migration 8-13–8-26
sending large amounts of 10-23
data curator 3-3
database
disk for 12-23
estimated size 12-27
logging in 6-7–6-13
logging out 6-7–6-13
modifying 6-8
outside a transaction 6-4
transaction mode and 6-7
pointers to objects in 12-24
preserving consistency 6-3
querying 5-4–5-19
DataCurator
privilege of 7-31
deadlocks, detecting 6-18

declaring temporary variables A-8
`decrement` (RcCounter) 6-28
default
  cluster bucket 12-6
  segment 7-6
defining
  error category 11-9
  error numbers 11-9
`definition` (Class) 13-4
`deleteObjectAt:` (Repository) 7-30
`deletePrivilege:` (UserProfile) 7-32
DependencyList 6-11
depth-first clustering 12-11
describing
  cluster buckets 12-7
`description:` (ClusterBucket) 12-7
`detect:` (Collection) 4-31, 5-19
determining
  a class's storage format 13-16
  class version 8-5
  current cluster bucket 12-6
  inheritance 13-4
  lock status 6-24
  object location on disk 12-14
  structure of class 13-4
Dictionary 4-2, 4-10
  for GemStone errors 11-10
  Globals 3-3
  internal structure 4-10
  keys 4-10
  shared 3-2–3-12
  UserGlobals 3-3
  values 4-10
`dictionaryNames` (UserProfile) 3-4
directory, examining 9-8
dirty locks 6-18
`disableSignaledAbortError` (System) 6-13
`disableSignaledGemStoneSession-Error` (System) 10-21
`disableSignaledObjectsError` (System) 10-10
disk
  access 12-2–12-15
  efficient use and number of cluster buckets 12-7
  location of database 12-23
  location of objects 12-2–12-15
  moving objects immediately to 12-11
  page for atomic objects 12-14
  pages cached from 12-24
  pages read or written per session 12-2
`do:` (Collection) 4-6
`do:` (RcQueue) 6-29
`do:` (SequenceableCollection) 4-18

E

efficiency
  creating collections 4-5
  data retrieval 12-2–12-15
  large arrays 4-22
  large strings 4-28
  locking collections 6-19
  searching collections 5-1–5-34
  selecting objects with streams 5-17
  Smalltalk DB execution 12-15–12-23
  sorting 5-34
  storage
    cache sizes and 12-26
Employee
  example class 4-38–4-40
  relation (table) 4-38, 5-2
  relation example 5-2
empty blocks A-20
empty paths
  index creation 5-23
  sorting 4-35
enableNotifyError 11-8
enableSignalAbortError 11-8
`enableSignaledAbortError` (System) 6-13
`enableSignaledGemStoneSession-Error` (System) 10-21

enableSignaledObjectsError 11-8
enableSignaledObjectsError (System) 10-10
ending a transaction 6-7–6-13
enumerating SequenceableCollections 4-18
enumeration protocol 4-6
environment variable in file specification 9-2
equality
  building indexes 5-22
  indexes 5-22
    creating 5-22
    re-creating within an application 5-15
  InvariantStrings 4-29
  operators 5-7
    redefining 5-8, 5-10–5-15
      rules 5-10
  queries 5-22
  SequenceableCollections 4-17
  strings 4-29
equalityIndexedPaths (Collection) 5-25
equalityIndexedPathsAnd-Constraints (Collection) 5-25
equalityIndexedPathsAndConstraints (Collection) 5-28
error 11-1–11-25
  flow of control and 11-9
  locking collections 6-20
  message, receiving from Stone 10-10, 10-20
  recursive 11-24
  RT_ERR_SIGNAL_COMMIT 10-10
  RT_ERR_SIGNAL_GEMSTONE-_SESSION 10-19, 10-20
  RT_ERR_SIGNAL_GEMSTONE_SESSION 10-20
  uncontinuable 11-24
  while comparing 5-8
  while compiling 13-6
  while creating indexes 5-29
  while executing operating system commands 9-9
  while migrating 8-11, 8-22
error category, defining 11-9
error dictionaries 11-2
  names of standard 3-5
error messages
  defining 11-2
error numbers
  compiler 13-6
  defining 11-2, 11-9
errors
  aborting B-1
  fatal B-2
ErrorSymbols 11-10, 11-16
  generic error in 11-10
examining
  directory 9-8
  symbol lists 3-4
Exception 11-5
exception 11-1–11-25
  context, defined 11-6
  raising 11-15, 11-16
  removing 11-23
  resignaling 11-21
  static, defined 11-6
  to receive intersession signals 10-20
  to receive notification of changes 10-11
exclusive locks 6-14
  defined 6-15
exclusiveLock: (System) 6-16
exclusiveLockAll: (System) 6-20
executing
  blocks A-19
  operating system commands 9-9
exists: (GsFile) 9-7
expressions
  combining A-13
  kinds A-4
  message A-9
  order of evaluation A-13
  syntax A-3
  value of A-18

extent
  cluster buckets and 12-4
  size of 12-5
extentForPage: (Repository) 12-14
extentId (ClusterBucket) 12-5
extentId: (ClusterBucket) 12-5

F

fatal errors B-2
FatalError 3-5
ff (GsFile) 9-5
file 9-2–9-9
  creating 9-3
  data in 9-10
  determining if open 9-7
  external to GemStone 6-11
  reading 9-5
  removing 9-7
  specifying 9-2
  temporary, for profiling 12-16
  testing for existence 9-7
  writing 9-5
fileName: (ProfMonitor) 12-16
fileSizeReport (Repository) 12-4
findDisconnectedObjects (Repository) 12-22
findFirst: (SequenceableCollection) 4-18
finding
  instances 8-8
findLast: (SequenceableCollection) 4-18
findPattern:startingAt: (String) 4-25
floating point number
  performance of 12-21
flow of control
  and blocks A-22
  changing with exceptions 11-9
  looping
    through a collection 4-6
format (Behavior) 13-16
formatting, code A-25
free variable
  defined 5-5
  in selection blocks 5-5
fully constrained paths 5-6

G

gatherResults (ProfMonitor) 12-17
Gem
  private page cache 12-24
    increasing size 12-26
    memory allocated for 12-25
  -to-Gem signaling 10-15–10-21
    with exceptions 10-20
gemprofile.tmp 12-16
gemSignalAction: (GSSession) 10-20
GemStone
  C Interface
    acquiring locks and 6-18
    linking your application 9-2
    logging in with 7-2
  caches 12-24–12-27
  hierarchy 2-12
  Smalltalk Interface
    acquiring locks and 6-18
    logging in with 7-2
GemStoneError 11-10
genericError 11-16
genericSignal:text: (System class) 11-16
Globals dictionary 3-3
grammar, Smalltalk A-27
group
  membership and performance 12-22
group: authorization: (Segment) 7-31
groupId
  instance variable for Segment 7-12
GsFile 9-2–9-9
GsSocket 9-12

H

handling errors 11-1–11-25
heap space for signals 10-19, 10-22
`hierarchy` (Class) 13-4
host machine
  RAM 12-27

I

identification, user 7-2
identifying
  object 11-5
  session 10-20
identity
  indexes 5-20
    creating 5-21
  InvariantString 4-29
  literal strings 4-29
  operator 5-7
  queries 5-6, 5-20
  sets 4-29
  strings 4-29
identity collection 4-37
IdentityBag 4-30–4-37
  accessing elements 4-31
  adding to 4-31
  nil values 4-30
  sorting 4-33
`identityIndexedPaths` (Collection) 5-25
IdentitySet 4-37–4-42
  nil values 4-30
implicit indexes 5-24
`includesSelector:` (Behavior) 13-7
`increment` (RcCounter) 6-28
indexed associative access 5-1–5-34
  comparing strings 5-10
indexed instance variables 4-8
indexing 5-1–5-34
  atomic objects 5-24
  authorization and 5-28
  automatically 5-24
  cluster buckets and 12-8
  concurrency control and 6-10–6-11
  creating equality index 5-22
  creating identity index 5-21
  equality 5-22
  errors while 5-29
  identity 5-20
  implicitly 5-24
  inquiring about 5-25
  keys 5-20
  locking and 6-22
  migration and 8-12
  NaN 5-30
  nil 5-30
  not preserved as passive object 9-10
  notify set and 10-11, 10-13
  on empty paths 5-23
  performance and 5-28
  RcBag and 6-29
  RcHashDictionary and 6-27
  re-creating equality, for user-defined operators 5-15
  removing 5-26
  sorting 5-34
  specifying 5-20
  structure 5-3, 5-30
    conflict on 6-11
  transactions and 5-30
  transferring to new collection 5-26
  updating indexed collections 5-27
IndexList 6-11
inheritance
  and constraints 2-6
  defined 2-12
  determining 13-4
inquiring
  about indexes 5-25
  about notify set 10-8

insertDictionary:At: (UserProfile) 3-6
inserting
  in a SequenceableCollection 4-15
  symbol list dictionaries 3-6
inspecting objects 3-5
instance
  finding 8-8
instance variables
  changing constraints of, and migration 8-20
  clustering 12-9
  constraining 2-5
  creating indexes on 5-20
  indexed 4-8
  migration and 8-13–8-26
  named
    in collections 4-2
  unordered 4-2
    constraining 4-8
    determining constraints of 4-10
    in Bags 4-30
  write set 6-10
instSize (Behavior) 13-16
instVarMapping: (Object) 8-17
instVarNames (Behavior) 13-13
Integer
  performance of 12-21
interfaces 6-18, 10-20
  linkable vs. remote 12-22
  returning control to, after error 11-5
interpreter
  halting because of authorization problem 7-16
  halting while executing operating system command 9-9
  method context in 11-6
intersession signal
  with exceptions 10-20
interval, sampling, for profiling 12-16
interval: (ProfMonitor) 12-16
inTransaction (System) 6-8
invalidElementConstraintWhenMigratingInto: for: (Object) 8-24
invariant classes 4-22
invariant objects
  changing 8-2
InvariantArray 4-22
InvariantString
  comparing 4-29
  identity 4-29
isBytes (Behavior) 13-16
isIndexable (Behavior) 13-16
isInvariant (Behavior) 13-16
isNsc (Behavior) 13-16
isPointers (Behavior) 13-16
iteration 4-6
itsOwner (instance variable for Segment) 7-12
itsRepository (instance variable for Segment) 7-12

K

kernel objects
  clustering methods 12-5
  clustering source code 12-5
  comparing 5-7
key
  dictionary 4-10
  sorting on secondary 4-36
keyword messages A-12, A-14
  maximum number of arguments A-12

L

lf (GsFile) 9-5
linkable interface 12-22
listing
  contents of directory 9-8
listInstances: (Repository) 8-8
listReferences: (Repository) 8-9

literal
  array 4-22, A-7
  blocks A-19
  character A-5
  number A-4
  symbol A-6
  syntax A-4
locks 6-8, 6-13–6-14
  aborting, effect of 6-23
  acquiring 6-16
  atomic objects and 6-16
  authorization for 6-16
  Boolean 6-16
  Character 6-16
  committing, effect of 6-23
  denial of 6-17
  difference between write and read 6-15
  dirty 6-18
  exclusive 6-14
    defined 6-15
  indexes and 6-22
  inquiring about 6-24–6-26
  limit on concurrent 6-14
  logging out, effect of 6-22
  manual transaction mode and 6-14
  nil 6-16
  on collections 6-19
  performance and 6-6
  read 6-13
    defined 6-14
  releasing upon commit 6-23
  releasing upon commit or abort 6-23
  removing 6-22
  shared 6-15
  SmallInteger 6-16
  types 6-14
  upgrading 6-21
  write 6-13
    defined 6-15
logging in 6-7–6-13
logging out 6-7–6-13
  effect on locks 6-22
loops 4-6

M

maintaining clustering 12-15
manual transaction mode 6-7–6-8
  defined 6-7
  locking and 6-14
`markForCollection` (Repository) 12-23
`maxClusterBucket` (System) 12-5
maximum number of
  arguments to a method A-12
  characters in a class name A-2
  cluster buckets for performance 12-7
  elements in a Bag 4-30
  elements in an unordered collection 4-30
memory 12-25
  changing allocations for caches 12-25
  increasing allocation for Gem private page cache 12-26
  increasing allocation for shared page cache 12-27
  increasing allocation for Stone private page cache 12-26
  increasing allocation for temporary object space 12-25
  of host machine 12-27
  requirements for passive objects 9-11
message
  arguments A-12
  binary A-12, A-13
  cascaded A-14
  expressions A-9
  keyword A-12, A-14
  privileged
    Segment 7-31
  sending, vs. path notation, performance of 12-21
  unary A-12, A-13

method
  accessing 13-2
  adding 13-2
  clustering for kernel classes 12-5
  compiling programmatically 13-5
  executing while profiling 12-16
  primitive 12-21
  recategorizing 13-10
  removing 13-4
  updating 13-2
`migrate` (Object) 8-9
`migrateFrom:instVarMap:` (Object) 8-18
`migrateInstances:to:` (Object) 8-10
`migrateInstancesTo:` (Object) 8-10
`migrateTo:` (Object) 8-7
Migrating
  indexed collection 5-27
migrating
  all instances of a class 8-10
  authorization errors and 8-13
  changed instance variable constraints and 8-20
  collection of instances 8-9
  errors during 8-11–8-13, 8-22
  indexed instances 8-12
  instance variable values and 8-13–8-26
  instances 8-7–8-26
  preparing for 8-8
  self 8-11
migration destination
  defined 8-7
  ignoring 8-10
`millisecondsToRun:` (System) 12-18
mixed-mode arithmetic 12-21
mode of transactions 6-7
modeling 1-3
modifying, *see* changing 8-1
`monitorBlock:` (ProfMonitor) 12-16
monitoring Smalltalk DB code 12-15
MonthNames 3-5
`moveMethod:toCategory:` (Behavior) 13-10
`moveToDisk` (Object) 12-11
`moveToDiskInBucket:` (Object) 12-11
moving
  methods between categories 13-10
  objects among segments 7-13
    authorization, indexing and 5-28
  objects on disk 12-15
  objects to disk immediately 12-11

N

named instance variables
  in collections 4-2
names
  of new versions 8-3
  variable A-7
NaNs
  and indexed queries 5-30
  sorting 4-35
Native Code 12-28
  enabling 12-28
  limitations of 12-29
  parameters 12-28
`nativeLanguage` (System) 11-3
`nativeLanguage:` (System) 11-3
`new` (ClusterBucket) 12-6
`newForExtent:` (ClusterBucket) 12-7
`newInRepository:` (Segment class) 7-31
`next` (RangeIndexReadStream) 5-17
`nextPutAll:` (GsFile) 9-5
nil 10-7
  comparing 5-8
  defined A-8
  in UnorderedCollection 4-30
  locking and 6-16
  segment of 7-14
  selection 5-30
nonsequenceable collection
  searching efficiently 5-1–5-34
notifiers 10-3

notify set
- adding objects 10-8
- asynchronous error for 11-8
- clearing 10-8
- defined 10-3
- indexing and 10-11, 10-13
- inquiring about 10-8
- permitted objects in 10-7
- removing objects 10-8
- size of 10-8

notifying user of changes 10-2–10-11
- by polling 10-12
- frequent changes 10-11
- improving performance 10-21

`notifySet` (System) 10-8
null values
- in large strings 4-28
- in new strings 4-24

Number
- literal A-4

`numberOfExtents` (Repository) 12-14

O

object
- average size 12-27
- change notification 10-2
- copying 12-22
- identifying 11-5
- local to application 6-12, 6-13
- moving 12-15
- moving among segments 7-13

object identifier, reporting in errors 11-5
object table 12-24
- cache for 12-27

operand
- defined 5-6
- selection block predicate 5-6–5-8

operating system
- accessing from Smalltalk 9-1
- executing commands from Smalltalk 9-9
- sockets 9-12

operator
- selection block predicate 5-7

optimistic concurrency control 6-11
- when to use 6-5

optimized selectors A-10
optimizing 12-1–12-27
- arrays vs. sets 12-21
- complexity of blocks and 12-22
- copying objects and 12-22
- creating HashDictionary 12-21
- frequency of commits and 12-23
- hints 12-21–12-23
- integers vs. floating point numbers 12-21
- linked vs. remote interface 12-22
- mixed-mode arithmetic and 12-21
- primitive methods and 12-21
- reclaiming storage and 12-22
- Smalltalk DB code 12-15–12-23
- string concatenation and 12-22
- using path notation vs. message-sends 12-21

OR (in selection blocks) 5-9
order of evaluation for expressions A-13
`owner` (Segment) 7-15
owner authorization 7-15
owner, changing, of a segment 7-15
`owner:` (Segment) 7-15
`ownerAuthorization:` (Segment) 7-31

P

`page` (Object) 12-14
page cache
- clustering and 12-27
- estimating size available to a transaction

12-23
Gem private 12-24
memory for 12-25
increasing size 12-26, 12-27
shared 12-24
memory allocated for 12-25
Stone private 12-24
memory for 12-25
`pageReads` (System) 12-2
`pageWrites` (System) 12-2
parameters A-12
block A-20
partially constrained paths 5-6
Pascal
comparing arrays 4-20
`passivate` (Object) 9-11
`passivate: toStream:` (PassiveObject) 9-10
PassiveObject 9-10–9-12
memory and 9-11
restrictions on 9-10
security considerations of 9-10
sending through a socket 9-12
password 7-2
path A-17–A-18
constrained 5-6
containing collection 5-31–5-33
defined A-17
empty
in index creation 5-23
sorting 4-35
operating system 9-2
performance of, vs. message-sending 12-21
`pathName` (GsFile) 9-7
pattern-matching in strings 4-26
`peek` (GsFile) 9-6
performance 12-1–12-27
arrays vs. sets 12-21
cluster buckets and 12-7
complexity of blocks and 12-22
copying objects 12-22
creating HashDictionary 12-21
determining bottlenecks 12-15
frequency of commits and 12-23
group membership and 12-22
indexing and 5-28
integers vs. floating point numbers 12-21
linked vs. remote interface 12-22
locking and 6-6
mixed-mode arithmetic 12-21
of primitive methods 12-21
of signals and notifiers, improving 10-21
path notation vs. message-sends 12-21
profiling 12-15
reclaiming storage and 12-22
reducing conflict and 6-27
segment participation in too many group authorizations 12-22
string concatenation and 12-22
tuning cache sizes 12-24–12-27
`performOnServer:` (System) 9-9
polling
to receive intersession signal 10-16, 10-20
to receive notification of changes 10-12
pool dictionaries
accessing 13-13
portability among versions 8-18
PositionableStream 4-43
precedence rules A-13
predicate
defined 5-5
in selection blocks 5-5
operators 5-7
terms 5-6
primitive methods 12-21
printable strings, creating with Stream 4-46
printing
strings 4-46

printOn: (Stream) 4-46
privilege
  changing 7-31
  defined 7-31
  garbage collection 12-23
process, spawning 9-9
profileOff (ProfMonitor) 12-18
profileOn (ProfMonitor class) 12-18
profiling
  report 12-19
  Smalltalk DB code 12-15
ProfMonitor 12-15–12-21
  method tally 12-16
  sampling interval 12-16
  temporary file for 12-16
pseudovariables 12-22, A-8
  nil A-8
  self A-9
  super A-9
  true A-9
PublishedSegment 3-12

Q

query 5-4–5-19
  Boolean operators in 5-9
  equality 5-22
  identity 5-20
  language 5-5

R

radix representation A-5
raising exceptions 11-15, 11-16
random access to SequenceableCollections 4-42
RangeIndexReadStream 5-17
RcBag 5-16, 6-6, 6-26, 6-29
  converting from bag 5-16
  indexing and 6-29
  methods implemented, compared to Bag 6-29
RcCounter 6-6, 6-26, 6-27–6-28
RcHashDictionary 6-6, 6-26, 6-31
  indexing and 6-27
RcQueue 6-6, 6-26, 6-29–6-30
  order of objects 6-30
  reclaiming storage from 6-30
read locks
  defined 6-14
  difference with write 6-15
read set 6-4, 6-5
  assigning object to segment 6-4
  indexing and 6-4
read/write authorization 7-10
read/write conflict
  defined 6-5
  reduced conflict classes and 6-27
  transaction conflict key 6-9
reading
  files 9-5
  in transactions 6-3
  outside a transaction 6-4
  SequenceableCollection 4-42
  with locks 6-13
readLock: (System) 6-16
readLockAll: (System) 6-20
ReadStream 4-43
receiving
  error message from Stone 10-10, 10-20
  intersession signal 10-19
    by polling 10-20
    with exceptions 10-20
  notification of changes 10-10–10-11
    by polling 10-12
    with exceptions 10-11
  signals by automatic notification 10-16
reclaiming storage 6-12, 12-22
  from temporary object space 12-24
  RcQueues and 6-30
  removing segments and 7-30
recomputeIndexSegments (AbstractBag) 5-28

recursive
- clustering 12-10
- errors 11-24

redefining
- classes 8-2–8-26
  - naming 8-3
- equality operators 5-8, 5-10–5-15
  - rules 5-10

reduced conflict class 6-26–6-31
- collections returned by selection 5-16
- performance and 6-27
- storage and 6-26
- temporary objects and 12-25
- when to use 6-27

`reject:` (Collection) 4-7, 5-19
`reject:` (IdentityBag) 4-7
relations 4-38
- Bags and Sets as 5-2

remote interface 12-22
- file access and 9-2

`remove` (Exception) 11-23
`remove:` (RcBag) 6-29
`remove:` (RcQueue) 6-29
`removeAllFromNotifySet:` (System) 10-8
`removeCategory:` (Behavior) 13-10
`removeClientFile:` (GsFile) 9-7
`removeDictionaryAt:` (UserProfile) 3-8
`removeFromCommitOrAbortReleaseLocksSet:` (System) 6-24
`removeFromCommitReleaseLocksSet:` (System) 6-24
`removeFromNotifySet:` (System) 10-8
`removeLock:` (System) 6-22
`removeLockAll:` (System) 6-23
`removeLocksForSession` (System) 6-23
`removeObjectFromBtrees` (Object) 5-15
`removeServerFile:` (GsFile) 9-7
`removeValue:` (Repository) 7-31

removing
- category 13-10
- exception 11-23
- files 9-7
- indexes 5-26
- locks 6-22
- method 13-4
- objects from notify set 10-8
- segments 7-30
- symbol list dictionaries 3-6, 3-8

`renameCategory:to:` (Behavior) 13-10
renaming a class 8-3
reordering symbol lists 3-6
repeating
- blocks A-23
- conditionally A-23

`report` (ProfMonitor) 12-17
reporting
- errors to user 11-1
- performance profile 12-19

reserved selectors A-10
`resignal:number:args:` (Exception) 11-21
resignaling another exception 11-21
resolving symbols 3-2–3-12
retaining data during migration 8-13–8-26
retrieving data quickly 12-2–12-15
return character
- in exception handler 11-18

returning values A-18
- from exceptions 11-18

`reverseDo:` (SequenceableCollection) 4-18
RT_ERR_SIGNAL_ABORT 6-12
RT_ERR_SIGNAL_COMMIT 11-8
rtErrCommitAbortPending 11-25
rtErrHardBreak 11-25
rtErrMessageBreakpoint 11-25
rtErrMethodBreakpoint 11-25
rtErrStackLimit 11-25
rtErrStep 11-25
rtErrUncontinuable 11-25
RuntimeError 3-5

S

sampling interval for profiling 12-16
saving
  data 9-10–9-12
  objects 6-11
schema 8-2
scientific notation A-5
`scopeHas:ifTrue:` (Behavior) 13-13
searching
  collections *see also* indexed associative access 4-7, 5-3
  protocol 5-3
  SequenceableCollection 4-18
secondary keys, sorting on 4-36
security
  locking and 6-16
  passive objects and 9-10
Segment
  assigning ownership 7-15
  changing before committing transaction 7-6
  default 7-6
  defined 7-3
  group authorization and performance 12-22
  moving objects 7-13
    authorization, indexing, and 5-28
  not preserved as passive object 9-10
  ownership 7-15
  predefined 7-12
  privileged messages 7-31
  published (defined) 3-12
  removing 7-30
segment
  authorization of 7-9
`segment` (Object) 7-13, 7-30
`select:` (Bag) 5-3
`select:` (Collection) 4-7, 5-3–5-10
`select:` (IdentityBag) 4-7
`selectAsStream:` (Collection) 5-17
  limitations of 5-18
selecting elements of a collection 4-7
selection block 5-4–5-15
  Boolean operators in 5-9
  collections returned 5-16
  comparing 5-7
  defined 5-3
  predicate
    comparing and 5-10
    defined 5-5
    free variables and 5-5
    operands 5-6–5-8
    operators 5-7
  streams returned 5-16
selection, conditional A-22
selector
  optimized A-10
  reserved A-10
`selectors` (Behavior) 13-7
`selectorsIn:` (Behavior) 13-10
self 12-22
  migrating 8-11
sending
  large amounts of data 10-23
  signal 10-17–10-21
  signal to another Gem session 10-18–10-20
`sendSignal:` (System) 10-19
`sendSignal:to:withMessage:` (System) 10-19
SequenceableCollection 4-13–4-30
  accessing 4-14
  accessing with streams 4-42
  adding to 4-14, 4-15
  comparing 4-17
  enumerating 4-18
  equality of 4-17
  inserting in 4-15
  searching 4-18
  updating 4-14
server, defined 9-2

session 6-28
  communicating between 10-2–10-23
  identifying 10-20
  pages read or written 12-2
  private page cache 12-24
  signaling all current 10-18
Set
  as relation 5-2
  identity 4-29
  nil values 4-30
  performance of 12-21
shallow copy 4-18
shared
  dictionaries 3-2–3-12
  locks
    defined 6-15
  page cache 12-24
    clustering and 12-27
    estimating size available to a transaction 12-23
    increasing size 12-27
    memory allocated for 12-25
`sharedPools` (Behavior) 13-13
sharing objects 3-2–3-12
shell script 9-9
signal 10-15
  receiving 10-19
    by polling 10-16, 10-20
  sending 10-17–10-21
  to abort, from Stone 6-12
`signal:args:signalDict:` (System) 11-2
`signaledAbortErrorStatus` 6-13
`signaledGemStoneSessionError-Status` (System) 10-21
`signaledObjects` (System) 10-11
`signaledObjectsErrorStatus` (System) 10-11
`signalFromGemStoneSession` (System) 10-19, 10-20
signaling
  all current sessions 10-18
  another session 10-18
    asynchronous error for 11-8
    by polling 10-20
  errors 11-2
  Gem-to-Gem 10-15–10-21
  improving performance 10-21
  order of receiving 10-19
`size` (RcQueue) 6-29
`size:` (Object) 4-20
`sizeOf` (GsFile) 9-7
`skip:` (GsFile) 9-6
SmallInteger 10-7
  locking and 6-16
  segment of 7-14
Smalltalk
  BNF syntax for A-27
  syntax A-1–A-26
socket 9-12–9-13
  sending passive objects through 9-12
sockets 10-21
sorting
  constrained collections 4-33
  empty paths 4-35
  indexing 5-34
  NaN 4-35
`sortWith:` (IdentityBag) 4-37
source code clustering 12-5
`sourceCodeAt:` (Behavior) 13-7
spacing in Smalltalk programs A-25
spawning a subprocess 9-9
special
  objects
    clustering and 12-10
    disk page of 12-14
    indexing and 5-24
    locking and 6-16
  selectors A-10
specifying files 9-2
`spyOn:` (ProfMonitor) 12-16
Stack (example class) 4-21

stack overflow 11-24
starting a transaction 6-7–6-13
`startMonitoring` (ProfMonitor) 12-17
state transition diagram of view 6-3
statement
  assignment A-9
  defined A-2
static exception
  defined 11-6
stdout 9-9, 10-19, 10-22
Stone private page cache 12-24
  increasing size 12-26
  memory allocated for 12-25
`stopMonitoring` (ProfMonitor) 12-17
storage
  format
    determining a class's 13-16
  reclaiming 6-12, 12-22
    from temporary object space 12-24
    RcQueues and 6-30
    removing segments and 7-30
  reduced conflict classes and 6-26
Stream 4-42–4-46
  on a collection 4-43
  returned by selection blocks 5-16
  to create printable strings 4-46
String 4-24–4-29
  comparing 4-29
    using associative access 5-10
  concatenating 4-27, 12-22
  identity 4-29
  large, and efficiency 4-28
  literal A-6
  pattern matching 4-26
  searching and comparing (table) 4-25
  using Streams to build 4-46
subclass, defined 2-12
subprocess, spawning 9-9
super 12-22
superclass
  defined 2-12
Symbol 3-2–3-12, 4-29
  determining symbol list for 3-11
  literal A-6
  resolving 3-2–3-12
  white space in A-6
symbol list 3-2–3-10, 8-5
  examining 3-3, 3-4
  order of searches 3-5
  removing dictionaries from 3-8
  reordering 3-6
symbol list dictionaries
  creating 3-6
  inserting 3-6
  removing 3-6
SymbolDictionary 4-12
  used to define errors 11-2
symbolList
  update from GsSession 3-10
symbolList instance variable of class UserProfile 3-2
`symbolResolutionOf:` (UserProfile) 3-11
syntax of Smalltalk A-1–A-26
System 6-9, 6-16
system administrator
  setting configuration parameters 12-25
SystemRepository
  not preserved as passive object 9-10
SystemSegment 7-14
SystemSegment (predefined system object)
  defined 7-12
SystemUser (instance of UserProfile)
  and SystemSegment 7-12
SystemUser, privileges of 7-31

T

tally of methods executed while profiling 12-16
temporary object space 12-24
  increasing size 12-25
  memory allocated for 12-25
temporary objects 10-7

temporary variables A-8
declaring A-8
term
predicate, conjoining 5-9
predicate, defined 5-6
selection block predicate 5-6
Topaz
logging in with 7-2
viewing symbol list dictionaries in 3-5
tracer 12-2
transaction 6-1
aborting 6-11
views 6-11
automatic mode 6-7
defined 6-7
committing
after changing segments 7-6
after recomputing index segments 5-28
moving objects to disk 12-11
performance 6-27, 12-23
conflict keys (table) 6-9
creating indexes in 5-30
defined 6-7
ending 6-7–6-13
failing to commit 6-10
manual mode 6-7–6-8
defined 6-7
locking 6-14
mode 6-7–6-13
modifying database 6-7
read set 6-5
reading in 6-3
reading outside 6-4
starting 6-7–6-13
updating views 6-12
when to commit 6-2
to optimize performance 12-23
write set 6-5
writing in 6-4
writing outside 6-4
`transactionConflicts` (System) 6-9
`transactionMode` (System) 6-7
`transactionMode:` (System) 6-7
transferring indexes 5-26
tuples 4-38
type-checking of instance variables 2-5

U

unary messages A-12, A-13
uncontinuable errors 11-24
uniqueness and dictionaries 4-10
UNIX commands, executing from Smalltalk 9-9
UNIX process, spawning 9-9
unordered collection
maximum size 4-30
unordered collections 4-30–4-42
unordered instance variables 4-2
constraining 4-8
determining constraints of 4-10
in Bags 4-30
UnorderedCollection 4-30
updating
indexed structures 5-27
method 13-2
views and 6-12
upgrading locks 6-21
user ID 7-2
user-defined class
redefining equality operators 5-8, 5-10–5-15
rules 5-10
UserGlobals 3-3
UserProfile
creating 7-3
establishing login identity 7-2
group membership and performance 12-22
native language variable used for errors 11-2
not preserved as passive object 9-10
symbol lists and 3-2

V

value
  dictionary 4-10
  returning A-18
`value` (block) A-19
`value` (RcCounter) 6-28
variables
  accessing 13-3, 13-13
  constraining instance 2-5
  creating indexes on instance 5-20
  free, in selection blocks 5-5
  global
    clustering 12-5
  instance
    clustering 12-9
    indexing and write set 6-10
  limits on length A-7
  names A-7
    case of A-7
    determining which class defines 13-13
  retaining values during migration 8-13–8-26
  temporary A-8
  type of 2-5
versioning classes 8-2–8-3
  defined 8-3
  reusable code and 8-18
view 6-8
  aborting a transaction 6-11
  defined 6-7
  invalid 6-12
  transition diagram 6-3
  updating a transaction 6-12
visibility of modifications 6-8

W

WeekDayNames 3-5
`whichClassIncludesSelector:` (Behavior) 13-7
`whileFalse:` (Boolean) A-23
`whileTrue:` (Boolean) A-23
white space in Smalltalk programs A-25
wild card character 4-26
  in file specification 9-2
workspace 3-5
  GemStone 3-5
world authorization 7-11
`worldAuthorization:` (Segment) 7-31
write locks
  defined 6-15
  difference with read 6-15
write set 6-4, 6-5
  assigning object to segment 6-4
  indexing and 6-4
  instance variables 6-10
write/write conflict
  defined 6-5
  reduced conflict classes and 6-27
  transaction conflict key 6-9
  while overriding a method 6-10
`writeLock:` (System) 6-16
`writeLockAll:` (System) 6-20
`writeLockAll:ifInComplete:` (System) 6-20
WriteStream 4-43
write-write conflict 6-5
writing
  files 9-5
  in transactions 6-4
  outside a transaction 6-4
  SequenceableCollection 4-42
  with locks 6-13