GemStone®

Programming Guide for GemStone/S 64 Bit

Version 3.2

April 2014

INTELLECTUAL PROPERTY OWNERSHIP

This documentation is furnished for informational use only and is subject to change without notice. GemTalk Systems, LLC, assumes no responsibility or liability for any errors or inaccuracies that may appear in this documentation.

This documentation, 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 express written authorization from GemTalk Systems.

Warning: This computer program and its documentation are protected by copyright law and international treaties. Any unauthorized copying or distribution of this program, its documentation, or any portion of it, may result in severe civil and criminal penalties, and will be prosecuted under the maximum extent possible under the law.

The software installed in accordance with this documentation is copyrighted and licensed by GemTalk Systems 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.

Use, duplication, or disclosure by the Government is subject to restrictions set forth in the Commercial Software - Restricted Rights clause at 52.227-19 of the Federal Acquisitions Regulations (48 CFR 52.227-19) except that the government agency shall not have the right to disclose this software to support service contractors or their subcontractors without the prior written consent of GemTalk Systems.

This software is provided by GemTalk Systems, LLC and contributors “as is” and any expressed or implied warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose are disclaimed. In no event shall GemTalk Systems, LLC or any contributors be liable for any direct, indirect, incidental, special, exemplary, or consequential damages (including, but not limited to, procurement of substitute goods or services; loss of use, data, or profits; or business interruption) however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use of this software, even if advised of the possibility of such damage.

COPYRIGHTS

PATENTS

GemStone software is covered by U.S. Patent Number 6,256,637 “Transactional virtual machine architecture”, Patent Number 6,360,219 “Object queues with concurrent updating”, Patent Number 6,567,905 “Generational garbage collector with persistent object cache”, and Patent Number 6,681,226 “Selective pessimistic locking for a concurrently updateable database”. GemStone software may also be covered by one or more pending United States patent applications.

TRADEMARKS

VMware is a registered trademark or trademark of VMware, Inc. in the United States and/or other jurisdictions.

GemStone, **GemBuilder**, **GemConnect**, and the GemStone logos are trademarks or registered trademarks of GemTalk Systems, LLC., or of VMware, Inc., previously of GemStone Systems, Inc., in the United States and other countries.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Sun, **Sun Microsystems**, and **Solaris** are trademarks or registered trademarks of Oracle and/or its affiliates. **SPARC** is a registered trademark of SPARC International, Inc.

HP, **HP Integrity**, and **HP-UX** are registered trademarks of Hewlett Packard Company.

Intel, Pentium, and **Itanium** are registered trademarks of Intel Corporation in the United States and other countries.

Microsoft, **MS**, **Windows, Windows XP, Windows 2003**, **Windows 7**, **Windows Vista** and **Windows 2008** are registered trademarks of Microsoft Corporation in the United States and other countries.

Linux is a registered trademark of Linus Torvalds and others.

Red Hat and all Red Hat-based trademarks and logos are trademarks or registered trademarks of Red Hat, Inc. in the United States and other countries.

SUSE is a registered trademark of Novell, Inc. in the United States and other countries.

AIX, **POWER5**, **POWER6**, and **POWER7** are trademarks or registered trademarks of International Business Machines Corporation.

Apple, **Mac**, **Mac OS, Macintosh**, and **Snow Leopard** are trademarks of Apple Inc., in the United States and other countries.

Other company or product names mentioned herein may be trademarks or registered trademarks of their respective owners. Trademark specifications are subject to change without notice. GemTalk Systems cannot attest to the accuracy of all trademark information. Use of a term in this documentation should not be regarded as affecting the validity of any trademark or service mark.

GemTalk Systems
15220 NW Greenbrier Parkway
Suite 240
Beaverton, OR 97006

Preface

About This Manual

This manual describes the GemStone Smalltalk language and programming environment — a bridge between your application's Smalltalk code and the GemStone database.

Prerequisites

This manual is intended for users that are at least somewhat familiar with the Smalltalk programming language and with its programming environment.

You should have the GemStone system installed correctly on your host computer, as described in the *GemStone/S 64 Bit Installation Guide* for your platform.

Terminology Conventions

The term "GemStone" is used to refer to the server products GemStone/S 64 Bit and GemStone/S, and the GemStone family of products; the GemStone Smalltalk programming language; and may also be used to refer to the company, now GemTalk Systems, previously GemStone Systems, Inc. and a division of VMware, Inc.

Typographical Conventions

This document uses the following typographical conventions:

- Smalltalk methods, GemStone environment variables, operating system file names and paths, listings, and prompts are shown in `monospace` typeface.
- Responses from GemStone commands are shown in an `underlined` typeface.

Executing the Examples

This manual includes many examples, which are provided in the form of Topaz commands. These examples can be executed using either the Topaz command-line interface, or using tools such as GemBuilder for Smalltalk (GBS) or another graphical interface to the GemStone/S server.

GBS or other IDE tools provide browsers and related tools that make it easier to define classes and methods. The text of the GemStone Smalltalk code examples themselves (excluding the Topaz commands) is the same whichever way you enter it.

When using Topaz, you must include extra commands to begin and end an example. If needed, refer to the Topaz manual for instructions about entering and executing the text of the examples.

Other GemStone Documentation

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

- *Topaz Programming Environment* – describes Topaz, a scriptable command-line interface to GemStone Smalltalk. Topaz is most commonly used for performing repository maintenance operations.
- *GemBuilder for Smalltalk Users's Guide* – describes GemBuilder for Smalltalk, a programming interface that provides a rich set of features for building and running client Smalltalk applications that interact transparently with GemStone Smalltalk.
- *GemBuilder for C* – describes GemBuilder for C, a set of C functions that provide a bridge between your application's C code and the application's database controlled by GemStone.
- *System Administration Guide* – describes maintenance and administration of your GemStone/S system.
- *VSD User's Guide* – describes VSD, a graphical tool to examine statistics data files generated by the GemStone/S server

In addition, each release of GemStone/S 64 Bit includes *Release Notes*, describing changes in that release, and platform-specific *Installation Guides*, providing system requirements and installation and upgrade instructions.

A description of the behavior of each GemStone kernel class is available in the class comments in the GemStone Smalltalk repository. Method comments include a description of the behavior of methods.

Technical Support

Support Website

http://gemtalksystems.com/techsupport

GemTalk's Technical Support website provides a variety of resources to help you use GemTalk products:

- **Documentation** for released versions of all GemTalk products, in PDF form.
- **Downloads**, including current and recent versions of GemTalk products.
- **Bugnotes**, identifying performance issues or error conditions that you may encounter when using a GemTalk product.
- **TechTips**, providing information and instructions that are not in the documentation.
- **Compatibility matrices**, listing supported platforms for GemTalk product versions.

This material is updated regularly; we recommend checking this site on a regular basis.

Help Requests

You may need to contact Technical Support directly, if your questions are not answered in the documentation or by other material on the Technical Support site. Technical Support is available to customers with current support contracts.

Requests for technical assistance may be submitted online, by email, or by telephone. We recommend you use telephone contact only for more serious requests that require immediate evaluation, such as a production system down. The support website is the preferred way to contact Technical Support.

Website: http://techsupport.gemtalksystems.com

Email: techsupport@gemtalksystems.com

Telephone: (800) 243-4772 or (503) 766-4702

When submitting a request, please include the following information:

- Your name and company name.
- The versions of GemStone/S 64 Bit and of all related GemTalk products, and of any other related products, such as client Smalltalk products.
- The operating system and version you are using.
- A description of the problem or request.
- Exact error message(s) received, if any, including log files if appropriate.

Technical Support is available from 8am to 5pm Pacific Time, Monday through Friday, excluding GemTalk holidays.

24x7 Emergency Technical Support

GemTalk offers, at an additional charge, 24x7 emergency technical support. This support entitles customers to contact us 24 hours a day, 7 days a week, 365 days a year, for issues impacting a production system. For more details, contact GemTalk Support Renewals.

Training and Consulting

GemTalk Professional Services provide consulting to help you succeed with GemStone products. Training for GemStone/S is available at your location, and training courses are offered periodically at our offices in Beaverton, Oregon. Contact GemTalk Professional Services for more details or to obtain consulting services.

Contents

Chapter 6. Numeric Classes *83*

Chapter 7. Indexes and Querying *93*

Chapter 8. Transactions and Concurrency Control 129

Chapter 9. Object Security and Authorization 155

Chapter

1 Introduction to GemStone

This chapter introduces you to the GemStone system. GemStone provides a distributed, server-based, multi-user, transactional Smalltalk runtime system, with the ability to partition the application between client and server. GemStone provides enterprise-quality security, scalability, availability, and services for managing and monitoring the repository.

1.1 GemStone Overview

Multi-User

GemStone can support thousands of concurrent users, object repositories of hundreds of gigabytes, and sustained object transaction rates of hundreds of 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, leveraging shared memory and SMP.

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. Changes that 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 method execution privileges and object access privileges.

Programmable

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 or Java, 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.

Scalable

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, and objects exist only in a single user's image.

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 authorized users.

However, since the GemStone repository is accessed through disk caches, it is not limited in size by available memory. A GemStone repository can contain billions of objects, each with a unique object identifier (known as an OOP—object-oriented pointer).

Object Database

GemStone lets you model information in structures as simple or complex as application data requires. You can represent data objects in tables, hierarchies, networks, queues, or any other structure or nested combination of structures that is appropriate.

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 7, "Indexes and Querying."

Partition Between Client and Server

GemStone applications can access objects and run their methods from a number of languages, including Smalltalk, C, Java, or any language that makes C calls. 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, Java classes, and C functions. These language interfaces 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.

GemStone's interfaces 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 Smalltalk is a separate product, and includes documentation describing installation and use.

GemBuilder for Java
GemBuilder for Java also has two parts: a set of GemStone programming tools, and a

programming interface between the client application code and GemStone. GemBuilder for Java is a Java runtime package that provides a message-forwarding interface between a Java client and a GemStone server, allowing access to objects in a GemStone repository. GemBuilder for Java is distributed as a separate product, and includes documentation describing installation and use.

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. GemBuilder for C is distributed with the server product. For more information on GemBuilder for C, see the *GemBuilder for C Guide*.

GLASS/Seaside

GLASS – GemStone, Linux, Apache, Seaside, and Smalltalk – provides a Pharo-compatible GemStone Smalltalk framework to create and deploy desktop-like web applications. A GemStone Seaside image is a base GemStone image with additional classes and tools loaded. For more information, see seaside.gemtalksystems.com.

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

In addition to these interfaces, GemStone provides a command-line tool that allows you to interact with server objects, execute code, and perform limited scripting.

Topaz

Topaz is a GemStone programming environment that provides a scriptable command-line interface to GemStone Smalltalk. Topaz is most commonly used for performing repository maintenance operations. 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. For more information on Topaz, see the *Topaz Programming Guide*.

Connect to Outside Data Sources

While GemStone methods are all written in Smalltalk (except for a limited number of primitives), you may often want to call out to other logic written in C. GemStone provides server al ways to access external code from a GemStone session.

UserActions (C callouts from GemStone Smalltalk)

UserActions are similar to user-defined primitives in other Smalltalks. You can use GemBuilder for C to write these user actions, and add them to and execute them from GemStone Smalltalk. The tools supporting user actions are part of the GemStone kernel, and are documented in the *GemBuilder for C* manual.

Foreign Function Interface (FFI)

FFI classes with GemStone allow you to invoke functions in existing C libraries. The argument and return data types are defined within GemStone Smalltalk to conform to the C function definition. The FFI interface is part of the GemStone kernel, and is documented in this Programming Guide.

GemConnect (Access to Oracle database)

GemStone uses the User Action mechanism to build the GemConnect product, which provides access to relational database information from GemStone objects. GemConnect is fully encapsulated and maintained in the GemStone object server. GemConnect is distributed as a separate product, and includes documentation describing installation and use.

1.2 GemStone Services

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, or refusing to commit conflicting changes.

GemStone provides both optimistic and pessimistic approaches to managing concurrent transactions, and supports explicit object locking for read or write. To allow users to modify the same object in ways that do not actually conflict, such as two users adding to a collection, GemStone extents the Collection class hierarchy by providing reduced-conflict (Rc) classes that can be used

For more on transactions and reduced-conflict classes, See Chapter 8, "Transactions and Concurrency Control."

Login 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.

Login Authentication

Before users can access system resources, they must be authenticated. Logins can be done from any of the interfaces; in each case, GemStone requires a user ID and a password, and a corresponding UserProfile must exist in GemStone. Authentication of the user ID and password can be done using GemStone's encryption or by Lightweight Directory Access Protocol (LDAP). GemStone uses SRP and SSL to establish secure logins and certain types of interprocess connections. Authentication and login security features are described in the *System Administration Guide*.

Object-level Authorization

To control access to individual objects, GemStone provides object-level authorization. Authorization enforcement is implemented at the lowest level of basic object access to prevent users from circumventing the authorization checking. Read and write authorization can be granted to single objects or groups of objects, for single users or groups of users. See Chapter 9, "Object Security and Authorization."

User Privileges

GemStone defines a set of privileges for controlling the use of certain system services. Privileges determine whether the specific user is allowed to execute certain system functions, usually ones only performed by the system administrator. Privileges are described in the *System Administration Guide*.

Services To Manage the GemStone Repository

GemStone is capable of managing objects shared by thousands of users, running methods that access billions of objects, and handling queries over large collections of objects by using indexes. 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. These services are described in the *System Administration Guide*.

1.3 GemStone Smalltalk

GemStone Smalltalk is tailored to operate in a multi-user environment, with transaction throughput and client communication as chief considerations. GemStone's class library is designed for multi-user access to objects. 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.

With a limited number of exceptions, GemStone Smalltalk supports the ANSI Smalltalk standard.

No User Interface

Because GemStone is an object server, GemStone Smalltalk does not provide any classes for screen presentation or user interface development. Graphical user interfaces, including those for developing classes and methods as well as runtime user interfaces, are provided by the client application. The client application uses a GemBuilder interface or a web interface such as Seaside to communicate and interact with the GemStone server.

A significant part of programming with GemStone is designing the interactions between various client runtime systems and the GemStone classes, methods, and objects on the server.

GemStone Sessions

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.

A session can be integrated with the application into a single process, called a *linked* application. Each application can have only one linked session.

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. An application may have multiple RPC sessions running simultaneously with each other and a linked session.

System Management Classes

GemStone Smalltalk provides a number of classes that offer system management functionality.

- The class System, which has no instances, provides class protocol to manage the repository.
- The class Repository, which has a single instance named SystemRepository, provides protocol for data management functions, such as extent creation and access, backup and restore, and garbage collection.
- The class UserProfileSet, which has a single instance named AllUsers, provides protocol to create and manage users.

Monitoring your application

GemStone includes statmonitor and Visual Stat Display (SD) utilities, which allow you to monitor and record, and view statistics about your application performance. This allows precise tuning as well as detecting potential problems before they occur. GemStone also includes profiling classes that allow you to optimize and tune your Smalltalk code for maximum performance.

File In and File Out

GemStone 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 create a text representations of the binary objects, which can be written to a file and read into another repository.

Interapplication 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.

1.4 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. The following sections describe the main aspects of GemStone architecture.

Gem Process

For each login, a GemStone session is established with a Gem process. 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 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. A user application is always connected to at least one Gem, and may have connections to many Gems. Gems can be distributed on multiple, heterogeneous servers.

In addition to Gem Processes for user sessions, a running GemStone system includes a number of maintenance Gem processes. These system Gems include the GcGems, which handle the tasks of collecting objects that are no longer referenced and the SymbolGem, which centralizes the creation of unique, canonical symbols.

Stone Process

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

NetLDI

Most GemStone configurations will includes a network server process, known as a NetLDI (Network Long Distance Information). The NetLDI is responsible for starting up GemStone processes such as Gems, and coordinates startup when GemStone processes are needed on a node other than the one the Stone is running on.

Shared Page Cache

The shared page cache (SPC) provides efficient retrieval of objects from disk, and the ability for multiple Gems to access the same object. The SPC is a large, contiguous area of

shared memory that is shared by the Stone and each Gem process on that host. Memory is managed and allocated on pages within this shared memory. A cache is started on each machine that runs a Stone monitor, Gem session process, or linked application.

The SPC also contains buffers for communications between Gems and the Stone. The Shared Cache Monitor process initializes the shared memory cache, manages 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 without performance degradation.

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 collection of one or more extents.

Transaction Log

GemStone's transaction log provides complete point-in-time roll-forward recovery. The transaction log contents are composed by the Gem, and the Stone writes the tranlog using asynchronous I/O. Commit performance is improved through I/O reduction, because only log records need to be written, not many object pages. In addition, the object pages stay in memory to be reused. Transaction logs may be on file systems or on raw devices.

Chapter

2 Class Creation

The first thing you will want to do is create the classes that will implement your application. This chapter describes class creation protocol, including some special features that can apply to all instances of a class.

Subclass Creation
explains how to define new GemStone classes, class implementation formats and other ways classes can store data.

Creating Classes With Invariant Instances
describes how to make objects invariant.

Creating Classes with Special Cases of Persistence
explains how classes can be defined so that their instances or instance variables are not stored in the repository.

2.1 Subclass Creation

Almost every class in the GemStone system understands a message that causes it to create a subclass of itself.

Example 2.1

```
Object subclass: 'Animal'
   instVarNames: #('habitat' 'name' 'predator')
   classVars: #('AllAnimals')
   classInstVars: #('AllOfSpecies')
   poolDictionaries: #()
   inDictionary: UserGlobals
```

This subclass creation message establishes a name (`'Animal'`) for the new class and provides for three named instance variables (`'habitat'`, `'name'`, and `'predator'`), a class variable (`'AllAnimals'`), and a class instance variable (`'AllOfSpecies'`). The new class in installed in the symbolDictionary UserGlobals of the user who executes this

code. You may also include reference to poolDictionaries, if this is useful for your application. Pool dictionaries are included by value, not by name; in other words, you use the reference to the pool dictionary, not a String.

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 1024 characters.

There are a number of subclass creation methods. The first keyword (in the example above, `subclass:`) defines the implementation format — more on this in the next section. Subclass creation methods with additional keywords are provided to provide other information to use when creating the class.

Some GemStone server classes cannot be subclassed. This is an attribute of the class. Execute *class* `subclassesDisallowed` to determine if a specific class can be subclassed.

Implementation Formats

Objects typically encapsulate data and behavior. The behavior is defined as methods on a class and the data is stored in the object. The data may be stored in named instance variables, indexed instance variables (Collection elements), or by value in specialized internal structures.

The implementation format refers to how the basic structure of the objects are defined by the class, which is done when the class is created. Implementation may be inherited from the superclass, or by using specific subclass creation methods you can specify the implementation format of the class.

Non-Indexable objects

Many types of objects have named instance variables, but no indexable variables. Objects may have up to 255 named instance variables, which are referred to by name in the code for that class. This is the default format; subclass creation methods that begin with the `subclass:` keyword will create classes of this format, if another format is not inherited.

Indexable Objects

Indexable objects have a variable number of instance variables that are referenced by an Integer index. The number of an object's indexed instance variables can increase dynamically at run time, up to 2^{40}-1 (about a trillion). There are two general cases of indexable objects:

Pointer-format

Pointer-format indexable objects allow the instance variables to refer to any other object. Pointer-format objects may also have up to 255 named instance variables.

Subclass creation methods that create indexed classes with pointer objects begin with the keyword `indexableSubclass:`.

Byte-format

This format is used for objects with indexed instance variables that are specialized for storing byte values, SmallIntegers in the range 0...255. Byte-format objects may not have named instance variables.

Subclass creation methods that create byte indexable classes begin with `byteSubclass:`.

You may not create byte-indexable subclasses of pointer-indexable classes, nor vice-versa, nor can you create indexable subclasses of NSCs.

NonSequencableCollection (NSC)

These classes store data with neither names nor indexes. They are suited to applications in which access is by value, rather than by name or position. Classes with this format are subclasses of UnorderedCollection, and are the classes for which Indexes are implemented.

You cannot directly define classes with this format, although you can subclass from existing kernel classes. Subclasses of NSC classes may have named instance variables, but not indexed instance variables.

Special

Instances of a few small, self-contained, kernel classes, including Character, SmallInteger, SmallDouble, Boolean, and UndefinedObject, are encoded entirely in the object identifier. Special objects do not use up an object ID (i.e., are not in the object table), do not take up separate space in the repository (beyond the original reference itself), and equal values always compare as identical.

You may not create your own specials nor may you subclass existing special classes.

Class Variables and Other Types of Variables

The implementation formats defined in the last section define several types of instance variables. Class definitions also include the following variable types:

Class variables
A class variable is a variable whose name and value are shared by a class, all of its instances, its subclasses, and all of their instances. Both class and instance methods of the class and its subclasses can refer to the variable. You can think of these variables as falling somewhere between local and global in their scope.

Class instance variables
A class instance variable is a variable whose name and value are shared by a class, but not by its instances. Subclasses inherit the variable's name but *not* its value. Only class methods of a class and its subclasses can refer to class instance variables. Class instance variables are useful when a class and its subclasses need to share the same structure, but not the same value, for a variable.

Pool variables
The pool variables are an Array of SymbolDictionary instances that are searched when attempting to bind a variable name during instance method compilation. Pool variables come after class variables and before globals in precedence. They are typically used when methods in a number of classes share values.

For example, one could define a SymbolDictionary with a key of #'CR' and a value of (Character codePoint: 13). If this SymbolDictionary were included in the class definition as a pool dictionary, then instance methods in the class could use CR as a way to reference the value and make the code more readable.

Global variables
Global variables are not tied to a class. They may be entries in a SymbolDictionary referenced in the UserProfile's SymbolList.

Dynamic Instance Variables

In addition to the fixed instance variables, which are the same for every instance of that class, you may also add dynamic instance variables to most instances.

Dynamic instance variables are key/value pairs that are stored with the instance like other instance variables, but may be added to specific instances of a class an not to other instances, without changing the class definition.

You cannot add dynamic instance variables to invariant objects, nor to Specials, nor to classes or metaclasses.

The maximum number of dynamic instance variables that can be added to an object is 255. However, the maximum may be lower for classes with many instance variables, since an object cannot be changed to a large object by adding dynamic instance variables. so, more exactly, the actual limit for the number of dynamic instance variables is calculated:

```
(255 min: ((2034 - self class instSize) / 2)
```

To add a dynamic instance variable, set the value using:

anObject `dynamicInstVarAt:` *nameSymbol* `put:` *value*

For example, say you have an instance of Animal representing the Bald Eagle. Bald Eagles are an endangered species, so you might want to add the legal and conservation information to this instance, but not to other instances of Animals.

```
theBaldEagle dynamicInstVarAt: #legalStatus
   put: 'Bald and Golden Eagle Protection Act'.
```

You can check what dynamic instance variables have been defined for an object:

```
topaz 1> printit
theBaldEagle dynamicInstanceVariables
%
an Array
  #1 legalStatus
```

and retrieve the stored value for a dynamic instance variable:

```
topaz 1> printit
theBaldEagle dynamicInstVarAt: #legalStatus
%
Bald and Golden Eagle Protection Act
```

If the Bald Eagle was no longer protected and this information was no longer needed, you could remove the dynamic instance variable

```
theBaldEagle removeDynamicInstVar: #legalStatus
```

The name and data for dynamic instance variables are persisted in the repository like any other instance variable data. Dynamic instance variables allow you to add instance variables to instance of a class, without the need to migrate. However, dynamic instance variables are less efficient than named instance variables, and make for code that is more difficult to maintain.

Additional Class Creation Protocol

In addition to implementation format and variables, there are other features of classes that can be, or must be, defined when the class is created. These are provided via subclass creation methods with additional keywords.

The subclass creation methods follow the form in example Example 2.2.

Example 2.2

```
Object subclass: 'Animal'
   instVarNames: #('habitat' 'name' 'predator')
   classVars: #('AllOfSpecies')
   classInstVars: #('AllAnimals')
   poolDictionaries: #()
   inDictionary: UserGlobals
   newVersionOf: Animal
   description: 'Class describing Animals'
   options: #()
```

The `newVersionOf:` allows you to create a new class that has the same classsHistory as an existing class; this will be covered in detail in Chapter 10.

The `description:` keyword allows you to provide documentation as part of the class definition. You can also explicitly set the comment after the class has been created by using the `comment:` method. For example:

```
Animal comment: 'Class describing Animal, created for the Program-
mers Guide'.
```

The `options:` keyword allows you to specify a collection of symbols to defined specific features of the new subclass. The options can include any of these:

`#dbTransient`	See page 36 for details. This option cannot be used in combination with `#instancesNonPersistent` or `#instancesInvariant`
`#disallowGciStore`	For internal use
`#instancesInvariant`	All instances of this class will be made invariant as soon as they are committed. If any class is defined with instancesInvariant, all its subclasses must also have instancesInvariant. Cannot be used in combination with `#instancesNonPersistent` or `#dbTransient`
`#instancesNonPersistent`	See page 35 for details. This option cannot be used in combination with `#dbTransient` or `#instancesInvariant`

#logCreation	Log class creation, including expressions that are the same as an existing class and do not create a new class instance or version, to the gem log or linked topaz output using GsFile class>>gciLogServer:
#modifiable	If this symbol is included, the class remains is modifiable after creation. No instances can be created until you make the class unmodifiable by sending it the message immediateInvariant.
#noInheritOptions	If this symbol is included, it must be first, and in this case options are not inherited from the superclass nor from an existing version of the class. This applies to the options #subclassesDisallowed, #disallowGciStore, #traverseByCallback, #dbTransient, #instancesNonPersistent, and #instancesInvariant
#subclassesDisallowed	No subclasses of the newly created class are permitted.
#traverseByCallback	For internal use.

For more details on class creation protocol, refer to methods in the image.

Note that subclasses creation protocol including the keywords inClassHistory:, isInvariant:, constraints:, isModifiable:, and instancesInvariant: may still appear, but are deprecated. Methods including these keywords should not be used.

2.2 Creating Classes With Invariant Instances

For data that must not ever be changed, GemStone provides two ways to make objects invariant or unchangeable. These are object-level invariance, and class-level invariance.

Per-Object Invariance

Any object can be made invariant by sending it the message immediateInvariant (a method defined by class Object). This mechanism provides a form of write-protecting objects that is useful for maintaining the integrity of your database. Once immediateInvariant is sent to an object, no modifications can be made to any of the object's instance variables, nor can the size or class of the object be changed. The immediateInvariant message takes effect immediately, but can be reversed by aborting the transaction in which it was sent. Once the transaction has been committed, you cannot reverse the effect of this message. The message isInvariant returns true if the receiver is invariant; false otherwise.

Invariance for All Instances of a Class

In class-level invariance, the definition of the class specifies that all instances of the class are invariant. Such an instance can be modified only during the transaction in which it is created. When the transaction is committed, the instance becomes invariant and no further modifications can be made to any of its instance variables, nor can the size or class

of the object be changed. This mechanism is useful for supporting literals in methods and in other limited situations, but is generally more cumbersome than object-level invariance.

Class-level invariance can be specified during class creation by including the `#instancesInvariant` symbol in the `options:` keyword argument. You cannot also define the class with non-persistent instances (`#instancesNonPersistent`), nor with non-persistent instances variable data (`#dbTransient`).

The following example creates a subclass of Animal whose instances are invariant:

Example 2.3

```
Animal subclass: 'InvariantAnimal'
   instVarNames: #()
   classVars: #()
   classInstVars: #()
   poolDictionaries: #()
   inDictionary: UserGlobals
   options: (#instancesInvariant)
```

2.3 Creating Classes with Special Cases of Persistence

In some cases, you may want either objects or the instance variables of objects to not be persistent, that is, not be written to disk. For example, you may want to include session-dependent information that shouldn't be read by another session, or data that is bulky and can be recreated easily. There are several ways to handle this.

Non-Persistent Classes

You can define a class as having only non-persistent instances. This means that instances of this class cannot be committed, so you cannot include references to instances of non-persistent classes within a persistent data structure.

To create a class with non-persistent instances, in the `options:` keyword argument, include the symbol `#instancesNonPersistent`. You cannot also define the class with non-persistent instances variables (`#dbTransient`), nor with invariant instances (`#instancesInvariant`).

As discussed in Chapter 4, GemStone provides a class called KeySoftValueDictionary, which allows you to manage non-persistent objects that are large and take time to create, but can be recreated whenever needed from small, readily available objects (tokens).

You cannot commit instances of a non-persistent class. If you attempt to do so, GemStone issues an error that indicates whether the object's class or a superclass is non-persistent. (The non-persistent status of a class is inherited by all of its subclasses.)

To determine whether a class's instances are non-persistent, you can send the following message:

theClass `instancesNonPersistent`

This message returns true if the class is non-persistent, false otherwise.

To make all instances of a class non-persistent, send the message:

theClass `makeInstancesNonPersistent`

Similarly, send this message to make all instances of a class persistent:

theClass `makeInstancesPersistent`

To make all instances of a class (and all of its subclasses) non-persistent, even if the class is non-modifiable:

`ClassOrganizer makeInstancesNonPersistent:` *theClass*

Similarly, you can send this message to make all instances of a class persistent, even if the class is non-modifiable:

`ClassOrganizer makeInstancesPersistent:` *theClass*

DbTransient

Classes can also be defined as DbTransient. Instances of classes that are DbTransient can be committed — that is, there is no error if they are committed — but their instance variables are not written to disk. This is useful if you need to encapsulate objects that should not be persistent, such as semaphores, within object structures that do need to be persistent and shared.

To create a class with DbTransient instances, in the `options:` keyword argument, include the symbol `#dbTransient`. You cannot also define the class with non-persistent instances (`#instancesNonPersistent`), nor with invariant instances (`#instancesInvariant`).

When a data structure containing an instance of a DbTransient class is committed, the instance variables of the DbTransient object are written to the repository as nil. Whenever a DbTransient object is read into a session, all of its instance variables are nil.

Since DbTransient instances are stored only in memory, they are affected by the in-memory GC operations. (See “Managing VM Memory” on page 270. Also see Chapter 11 of the *System Administration Guide*.)

If memory becomes low, the transient objects may be stubbed out of memory. When needed, it is re-read from the repository. However, all the instance variables will be nil after a re-read. To prevent losing non-nil instance variable values, you should keep a reference to DbTransient instances in session state.

Since the DbTransient object will remain in memory while referenced from session state, the reference from session state should be removed when the DbTransient object is no longer needed, to avoid filling up memory and causing an out of memory error.

Note that while DbTransient objects are only committed once (on creation), and so do not normally cause concurrency conflicts, if they are clustered the object will be written (still with all instance variables nil), and could potentially cause a concurrency conflict.

To set a class so all instances are DbTransient, send:

aClass `makeInstancesDbTransient`

aClass must be a non-indexable pointer class. This will cause any instance of *aClass* to be DbTransient. The change takes place immediately.

The following message:

aClass `makeInstancesNotDbTransient`

will cause instances to be non-DbTransient, that is, allow instance variables to be written to disk.

Chapter

3 Resolving Names and Sharing Objects

This chapter describes how GemStone 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 GemStone Smalltalk allows users to share objects of any kind.

The Session-Based and UserProfile Symbol Lists
describes the mechanism that the GemStone 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

GemStone Smalltalk permits concurrent access by many users to the same data objects. For example, all GemStone 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 to your current session (see the following discussion).

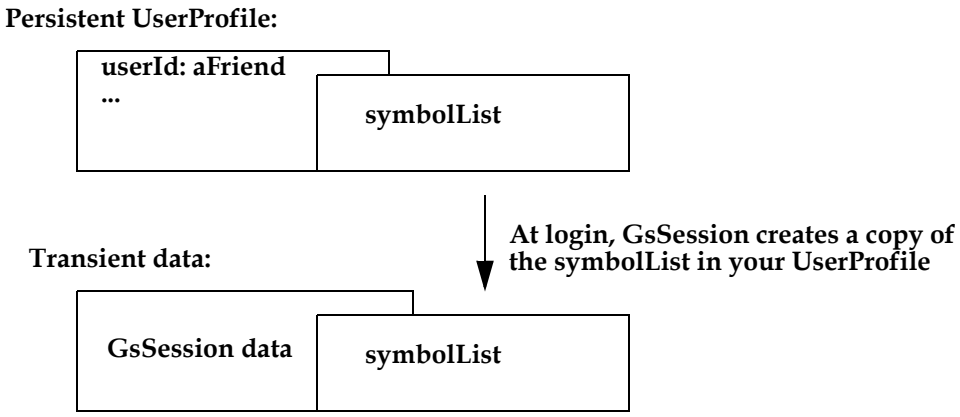
If GemStone cannot find a match for a name in one of these areas, you are given an error message.

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, 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. GemStone Smalltalk refers to this copy of the symbol list to find objects you name in your application. See Figure 3.1.

Figure 3.1 The GsSession symbolList — a copy of the UserProfile symbolList



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, nor are they subject to rollback after an abort.

Changes to the current session's symbolList do not affect the UserProfile symbolList. Thus, the UserProfile symbolList can 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?

In creating your UserProfile symbol list, the data curator adds SymbolDictionaries containing associations that define the names of all objects that the data curator 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 two dictionaries:

- A "system globals" dictionary called *Globals*. This dictionary contains some or all of the GemStone 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

```
topaz 1> printit
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 per-user application and application service objects.
- UserClasses
  Contains per-user class definitions, and is created by 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 dictionary:

- If you are using Topaz, execute some expression that returns the dictionary. Example 3.2 lists the dictionary keys. Alternatively, you could execute `UserGlobals` to examine all keys and values.

- If you are running GemBuilder for Smalltalk (GBS), select the expression `UserGlobals` in a GemStone workspace and execute `GS-Inspect it`.

Example 3.2

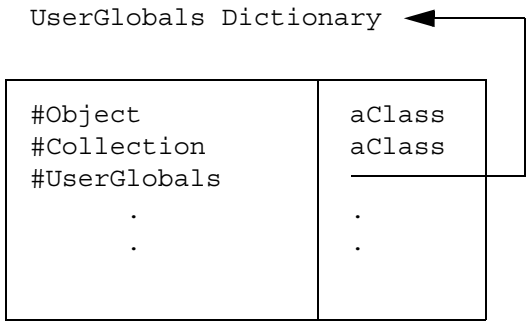
```
topaz 1> printit
UserGlobals keys
%
a SymbolSet
  ...
  #1 GcUser
  #2 UserGlobals
  #3 GsPackagePolicy_Current
  #4 PackageLibrary
  ...
```

If you examine all of your symbol list dictionaries, you'll see that most of the kernel classes are listed. In addition, there are global variables, both public and for internal use. For a detailed description of GemStone kernel objects, see Appendix D of the *System Administration Guide*.

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, GemStone Smalltalk finds only the first definition.

Figure 3.2 Self-Referencing Symbol Dictionary



Inserting and Removing Dictionaries from Your Symbol List

NOTE

To insert or remove a SymbolDictionary to/from your symbol list, you must have

the necessary system privilege. For details, see "User Accounts and Security" in the System Administration Guide.

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 Example 3.3, the new dictionary is inserted into the UserProfile symbolList and then updated in the current session.

Because the GemStone 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` (a class) 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`. This means that the GemStone object server will always find `MyCollection` in `MyClassDict`, not in `UserGlobals`.

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: #()
     classInstVars: #()
     poolDictionaries: {}
     inDictionary: myClassDict.
GsSession currentSession userProfile insertDictionary: myClassDict at: 1.

Object subclass: 'MyCollection'
     instVarNames: #('snakes' 'snails' 'tails')
     classVars: #()
     classInstVars: #()
     poolDictionaries: {}
     inDictionary: UserGlobals
```

Recall that the object server returns only the *first* occurrence found when searching the dictionaries listed by the current session's symbol list. When you subsequently refer to `MyCollection`, the object server returns only the version in `MyClassDict` (which you inserted in the first position of the symbol list) and ignores the version in `UserGlobals`. If you had inserted `MyClassDict` *after* `UserGlobals`, the object server would only find 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

Avoid redefining any kernel classes. 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 appropriate.

To remove a symbol dictionary, send your UserProfile the message `removeDictionaryAt:` *anInteger*, passing in the index of the dictionary you want to remove.

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 symbol list 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 can purposely set your current session symbol list to include new objects for testing.

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

`insertDictionary:` *aDictionary* `at:` *anIndex*
: This method inserts a Dictionary into the UserProfile symbol list at the specified index.

`removeDictionaryAt:` *anIndex*
: This method removes the specified dictionary from the UserProfile symbol list.

`symbolList:` *aSymbolList*
: This method replaces the UserProfile symbol list with the specified symbol list.

Each of these methods modifies the UserProfile symbol list. If the receiver is identical to "`GsSession currentSession userProfile`", the current session's symbol list 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.

In Example 3.5, the transient symbol list is copied 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 3.5

```
"Copy the GsSession symbol list to the UserProfile"
System myUserProfile symbolList:
   (GsSession currentSession symbolList copy).

"Check that the symbol lists are the same"
GsSession currentSession symbolList =
   System myUserProfile symbolList.

"Add a new dictionary to the current session"
GsSession currentSession symbolList add: SymbolDictionary new.

"Compare the two symbol lists; they should differ"
GsSession currentSession symbolList =
   System myUserProfile symbolList.

"Update 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 3.6

```
topaz 1> printit
System myUserProfile symbolResolutionOf: #Bag
%
2 Globals
    Bag  Bag
```

If *aSymbol* is defined in more than one dictionary, `symbolResolutionOf:` finds only the first reference.

To find out which dictionaries stores a name for an object and what that name is, send your UserProfile the message `dictionariesAndSymbolsOf:` *anObject*. This message returns an array of arrays containing the dictionaries in which *anObject* is stored, and the symbols which name that object in that dictionary.

Example 3.7 uses `dictionariesAndSymbolsOf:` to find out which dictionaries in the symbol list stores a reference to class DateTime.

Example 3.7

```
| anArray myUserPro |
myUserPro := System myUserProfile.

"Find first Dictionary containing DateTime"
anArray := (myUserPro dictionariesAndSymbolsOf: DateTime) first.
anArray at: 1.
aSymbolDictionary

"Get the name of the SymbolDictionary"
(anArray at: 1) keyAtValue: (anArray at: 1)
Globals
```

Note that `dictionariesAndSymbolsOf:` may return zero, one, or multiple dictionaries.

3.3 Using Your Symbol Dictionaries

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.

NOTE

To insert or remove a SymbolDictionary to/from your symbol list, or to make any changes to a UserProfile that is not your own, you must have the necessary system privilege. For details, see "User Accounts and Security" in the System Administration Guide.

Publishers, Subscribers and the Published Dictionary

The Published Dictionary, PublishedObjectSecurityPolicy, 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. You can use the Published dictionary to "publish" application objects to all users — for example, symbols that most users might need to access. The Published Dictionary is not used by GemStone classes; rather, it is available for application use.

The PublishedObjectSecurityPolicy is owned by the Data Curator and has World access set to none. Two groups have access to the PublishedObjectSecurityPolicy:

- Subscribers have read-only access.
- Publishers have read-write access.

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

For example, your system administrator might add each member of a programming team to the group Publishers. After completing the definition of a new class, a programmer could make the class available to colleagues by adding it to the Published dictionary. Because this dictionary is already in each user's symbol list, whatever you add becomes visible to users the next time they obtain a fresh transaction view of the repository. Using the Published dictionary lets you share these objects without having to put them in Globals, which contains the GemStone kernel classes, and without the necessity of adding a special dictionary to each user's symbol list.

Chapter

4 Collection and Stream Classes

The Collection classes are a key group of classes in GemStone Smalltalk. This chapter describes the common functionality available for Collection classes.

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

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

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

Sorting
describes the ways to sort elements in collections.

4.1 An Introduction to Collections

Instances of the Collection classes are specialized to manage an indeterminate number of objects as a group using unnamed instance variables. All instances of Collection subclasses support protocols for adding and removing elements (as long as the collection is not invariant), for iterating over the elements, and for testing the presence of an object. Collections can be classified by whether or not they maintain a specified order for their elements, whether or not key-based lookup is supported, and the kinds of objects they can reference.

Collections can be broadly classified into three categories:

- Access by Key — the Dictionary Classes

  Instances of AbstractDictionary subclasses do not support a specific order for their elements but do support storage and retrieval via the `at:put:` and `at:` messages, using arbitrary objects for an element's key. Subclasses of AbstractDictionary are specialized based on whether key-based lookup uses equality comparison or identity comparison, the type of key, and the type of value.

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

- Access by Position — the SequenceableCollection Classes

  Instances of SequenceableCollection classes maintain a specific order for their elements and support storage and retrieval via the `at:put:` and `at:` messages using an integer key (the one-based offset into the elements), analogous to an array with a numeric subscript in other programming languages.

  Byte-format classes such as ByteArray and String cannot have named instance variables. The other sequenceable collections can have named instance variables if you choose to define them.

- Access by Value — the UnorderedCollection Classes

  Instances of UnorderedCollection classes—also referred to as Non-Sequenceable Collections or NSCs—do not have a specific order for their elements, and do not support storage or retrieval via the `at:put:` and `at:` messages. Objects in these collections are accessed by iterating the collection. UnorderedCollections support indexes, which allow ordered iteration and fast key-based lookup.

  UnorderedCollections may have named instance variables.

Efficient Implementations of Large Collections

When you create a collection of more than about 2K pointer object or more than 16K byte objects, GemStone internally uses a sparse tree implementation to make more efficient use of resources. These are referred to as "Large Objects", and use internal classes such as LargeObjectNode. This behavior occurs in a manner that is transparent to you, and you handle Large Objects no differently than objects that are not large.

Protocol Common to All Collections

Collection classes understand common protocol, inherited from the abstract superclass Collection. 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 examples that follow provide a starting point for using Collections; review the methods and method comments in the image for more details.

Creating Instances

Collection classes respond to the 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 many Collection subclasses to create an instance that is pre-sized to hold *anInteger* elements. It's often more efficient to use `new:` than `new`, because a Collection created with `new:` need not expand repeatedly as you add new elements. This is particularly significant for large key-based Collections

where the hash must be computed for each element in the collection when the Collection base size changes. For very large collections, growing may require a large amount of temporary object memory to complete, with the risk of running out of memory.

Collections also define the instance creation message, `withAll:` *aCollection*, that creates a new instance of the receiver containing all of the objects stored in *aCollection*, and `with:`, `with:with:`, `with:with:with:`, and `with:with:with:with:`, which create a new instance of the receiver with 1, 2, 3 or 4 (respectively) specific elements.

Adding Elements

Collection defines for its subclasses two basic methods for adding elements: `add:`, which adds one element to the Collection, and `addAll:`, which adds several elements to the Collection at once.

Collection subclasses override these methods in order to control access to elements or to enforce an ordering scheme. Certain subclasses of Collection provide additional methods that add an element at a numbered positions or based on an arbitrary key.

Removing Elements

Collection defines for its subclasses several methods for removing elements: `remove:`, which removes one element to the Collection, and `removeAll:`, which removes a collection of elements from the Collection at once.

While it is an error to remove objects that are not in the collection, the methods `removeIfPresent:` and `removeAllPresent:` perform the same removals, but do not error if the specified objects or objects are not in the collection.

Enumerating

Collection defines several methods that enable you to loop through a collection's elements. 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 Array in this way:

```
UserGlobals
   at: #Virtues
   put: { 'humility' . 'generosity' . 'patience' }.
```

This defines a Array constructor - an Array that is created containing the three given literal strings.

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

Example 4.1

```
| 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
%
' generosity humility patience'
```

In addition to `do:`, Collection provides several specialized enumeration methods; the most common ones are `collect:`, `select:`, `detect:`, and `reject:`.

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 while you are enumerating it.

4.2 Collection Subclasses

This section describes the properties of Collection's concrete subclasses, and gives 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.

Dictionaries

Dictionary classes are subclasses of AbstractDictionary. The elements in a Dictionary collection are stored and accessed via a key; each key must be unique within that Dictionary. Depending on the specific subclass, the keys may be compared using equality or identity.

Dictionaries provide their special facilities by storing key-value pairs instead of simple, linear lists of objects. While some types of dictionaries are implemented as "a collection of Associations", the interface methods return results based on the logical contents, which are the values. Other, specialized protocol allows you to refer to the key or the value portions of the logical associations.

Internal Dictionary Structure

For performance reasons, the internal implementation of Dictionary classes varies. Instance of Dictionary itself consist of a collection of Association objects. KeyValueDictionary subclasses are implemented differently, as a sequence of keys and values, which may use CollisionBuckets to hold the actual values. IdentityDictionary is a

sequence of keys and Associations. All these dictionaries understand common protocol, regardless of implementation.

Dictionary

Dictionary class uses Associations to store the key/value pair. Dictionaries compare keys by equality, not by identity. If you need a dictionary that compares keys by identity, use IdentityDictionary, which is a subclass of KeyValueDictionary.

KeyValueDictionary

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

- IdentityKeyValueDictionary
- IntegerKeyValueDictionary
- StringKeyValueDictionary
- SymbolKeyValueDictionary
- IdentityDictionary
- SymbolDictionary

KeySoftValueDictionary

A KeySoftValueDictionary is a subclass of KeyValueDictionary that allows the virtual machine to remove entries as needed to free up memory.

Typically, you might use a KeySoftValueDictionary to manage non-persistent objects that are large and take time to create, but that can be recreated whenever needed from small, readily available objects (tokens). For example, you might create a KeySoftValueDictionary to serve as a cache to hold large, expensive objects that are needed repeatedly. Within that dictionary, the values would be the large calculated objects, and the keys would be the corresponding tokens. If your application needs a large, expensive object but does not find it in the KeySoftValueDictionary, you can create the object and add it to the cache so that it might be available the next time it is needed.

As memory fills up, the virtual machine might remove some objects from the cache. (Remember, the contents of the cache are non-persistent and can be recreated.) The virtual machine may remove keys and values from the KeySoftValueDictionary until adequate memory is available. For details about how to manage the number of KeySoftValueDictionary entries, see “Getting Rid of Non-Persistent Objects” on page 270.

Bear in mind the following:

- Entries are removed from a KeySoftValueDictionary only if there are no strong references to the entry’s value.
- If an entry in a KeySoftValueDictionary is cleared, all other entries that reference this value directly or indirectly will also have been cleared.
- Before generating an OutOfMemory error, the virtual machine removes all KeySoftValueDictionary entries that are eligible for removal.

- KeySoftValueDictionary entries are cleared during a mark/sweep operation, but are not cleared during a scavenge. For more about mark/sweep and scavenge operations, see the "Managing Growth" chapter of the *System Administration Guide*.
- A corresponding subclass, IdentityKeySoftValueDictionary, uses identity (rather than equality) comparison on keys. For details, see the image.
- A KeySoftValueDictionary frequently contains instances of SoftReference. Do not be tempted to confuse this with the notion of WeakReference found in many Smalltalk dialects; the two mechanisms are quite different.

SequenceableCollection

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. 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.

Adding and Removing Objects for SequenceableCollection

SequenceableCollection redefines `add:` so it puts the added object at the end of the receiver.There are additional methods for adding one or more objects to its instances at particular locations, and for removing one or more objects according to position, equality, or identity.

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. In order for two SequenceableCollections to be considered equal, the following conditions must be met:

- 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.

Copying SequenceableCollection

SequenceableCollection understands several copying message, including those that:

- return a sequence of the receiver's elements as a new collection
- copy a sequence of the receiver's elements into an existing SequenceableCollection
- copy elements from one SequenceableCollection into another, without faulting the contents into memory.

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

```
#('bear' 'tiger' 'turtle') copyFrom: 1 to: 2
%
an Array
#1 bear
#2 tiger
```

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

```
| numericArray |
numericArray := Array with: 55; with: 66 with: 77 with: 88.
numericArray replaceFrom: 2 to: 3 with: #( 1 2 3 4 5 )
   startingAt: 4.
numericArray
%
an Array
 #1 55
 #2 4
 #3 5
 #4 88
```

The advantage of using the method `replaceFrom:to:with: startingAt:` is that it does not fault the contents into memory, which can improve performance when working with very large collections. Of course, displaying the results as in the example also faults the objects into memory.

Also keep in mind that copies of SequenceableCollection, like most GemStone 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.

Array

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.

One of the most important differences between client Smalltalk arrays and a GemStone Smalltalk array is that GemStone arrays are extensible; you can increase the size of an array at any time. Sending `at:put:` will increase the size of the array, as long as the index is only one more than the current array size. Other protocol such as `addAll:` also increase the size while adding elements.

It's also possible for you to change the size without explicitly storing or removing elements, using the message `size:` inherited from class Object. When you lengthen an array with `size:`, the new elements are set to nil.

Creating Arrays

You are free to create an array with the inherited message `new` and let the array lengthen automatically as you add elements.

Arrays created with the `new` message initially allocate no space for elements. As you add objects to such an array, it must lengthen itself to accommodate the new objects. It's usually more efficient 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: 100.
```

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.

Literal Array and Array Constructors

Arrays can also be created in code without sending instance creation messages, by using literal array or array constructor syntax.

An Array literal is created with the following syntax:

`#(` *element1 element2 element3* `)`

When your code includes a statement like this, at compile time an invariant instance of Array is created.

An Array constructor is created at runtime, rather than at compile time, and it not invariant - it can be modified by later code. The syntax for Array constructors is:

`{` *element1* `.` *element2* `.` *element3* `}`

Array constructors are not part of the ANSI standard.

SortedCollection

SortedCollection is a type of SequenceableCollection in which the elements are ordered by a specific sort order, not by the order in which they were added or by the method used to add the element. You are not permitted to send `at:put:`, `addLast:`, or similar methods to a SortedCollection.

Each instance of SortedCollection is associated with its own sortBlock. The default block will sort elements that have a known sort order, such as alphabetic or numeric; the elements must be able to be compared using <=.

You can also define your own sortBlock, if you want elements ordered by some other criteria, such as the value of an instance variable.

For more on comparison, sorting, and sort blocks, see "Sorting" on page 62.

Example 4.2

```
| scrabbleWords |
scrabbleWords := SortedCollection sortBlock:
   [:a :b | a size < b size].
scrabbleWords add: 'able'; add: 'zebra'; add: 'jumper';
   add: 'yet'.
scrabbleWords
%
aSortedCollection( 'yet', 'able', 'zebra', 'jumper')
```

There is overhead in always keeping the collection sorted, so it usually more efficient to sort the elements only when you need them to be sorted for presentation. There are advantages to using a type of collection that is more suitable for managing the elements, then using methods such as `sortWithBlock:` to create a new Array with the original collection's elements in sorted order.

UnorderedCollection

Instances of UnorderedCollection store their elements in a private, implementation-defined order and explicit key-based access such as `at:` and `at:put:` are disallowed.

In all subclasses of UnorderedCollection, nil elements are disallowed. An UnorderedCollection will silently ignore attempts to add a nil element.

UnorderedCollection implements protocol for indexing, which allows for large collections to be queried and sorted efficiently. Chapter 7, "Indexes and Querying," describes the querying/sorting functions in detail. The following section describes the protocol implemented in UnorderedCollection's subclasses.

The most efficient way to handle very large collections is using UnorderedCollections and access the contents using GemStone indexes.

UnorderedCollections may also be referred to as Non-Sequenceable Collections (NSCs).

Bag and Set

The classes Bag and Set are some of the simplest UnorderedCollections. In these classes duplicate checking is done based on equality (rather than identity), and a Set will ignore attempts to add a equal element. A Bag will accept an equal item but will do so by increasing the count of the existing element. Thus, adding two equal but not identical objects will be treated as if the first object is present twice.

If the Bag or Set contains elements that are themselves complex objects, determining the equality is complex and therefore slower than you might have hoped. GemStone recommends using IdentityBag or IdentitySet for anything but the most simple unordered collections.

IdentityBag

IdentityBag has faster duplicate checking than Bag. Like a Bag, an IdentityBag is elastic and can hold objects of any kind.

To compare an object that is in 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.

The inherited messages `add:` and `addAll:` work much as they do with other kinds of collection, except, of course, that they are not guaranteed to insert objects at any particular positions. There's one other significant difference: if the argument to `addAll:` is an Array or OrderedCollection, the elements in the collection are not faulted into memory.

IdentityBag also defines a method that allow you to copy elements into a Collection (which must be a kind of Array or OrderedCollection) without faulting the contents into memory, using the message:

`replaceFrom:` *startIndex* `to:` *stopIndex* `with:` *aCollection* `startingAt:` *repIndex*

Example 4.3

```
| bagOfRodents |
bagOfRodents := IdentityBag withAll: #('beaver' 'rat'
   'agouti' 'chipmunk' 'guinea pig').
(Array new: 5) replaceFrom: 3 to: 5  with: bagOfRodents
   startingAt: 1.
 anArray( nil, nil, 'beaver', 'rat', 'agouti')
```

Accessing an IdentityBag's Elements

You'll generally 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 4.4

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

Removing Objects from an 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 4.5

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

```
anArray( 2, 4, 5)
```

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. Example 4.5 works correctly because the SmallInteger 2 has a unique identity throughout the system. By way of contrast, consider Example 4.6.

Example 4.6

```
| 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,
- 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 set union, set intersection, and set difference operators.

+ union, returning elements that are in either one, the other, or both.

- difference, returning elements that are in the receiver but not the argument.

* intersection, returning elements that are in both

Example 4.7

```
| pets rodents |
pets := IdentityBag with: 'dog' with: 'cat' with: 'gerbil'.
rodents := IdentityBag with: 'rat' with: 'gerbil' with: 'beaver'.
pets * rodents
%
 anIdentityBag( 'gerbil')

pets + rodents
%
 anIdentityBag( 'beaver', 'rat', 'gerbil', 'gerbil', 'cat', 'dog')

pets - rodents
%
 anIdentityBag( 'cat', 'dog')
```

There is one significant difference between these messages and set operators — IdentityBag's messages consider that either the receiver or the argument can contain duplicate elements. The method comment in the image provide more information about how these messages behave when the receiver's class is not the same as the class of the argument.

IdentitySet

IdentitySet is similar to IdentityBag, except that IdentitySet cannot contain duplicate (that is, identical) elements.

4.3 Stream Classes

Reading or writing a SequenceableCollection's elements in sequence entails some extra effort — you need to maintain an index variable so that you can keep track of which element you last processed. A Stream acts like a SequenceableCollection that keeps track of the index most recently accessed.

There are several concrete Stream classes. Class ReadStream is specialized for reading SequenceableCollections and class WriteStream for writing them; ReadWriteStream is also provided, for ANSI compatibility.

A stream provides 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.

Streams are often used for reading characters from strings or files, but any kind of collection can be used with a Stream, and any type of object can be in that collection.

PositionableStream and Position

PositionableStream, with its subclasses ReadStream and WriteStream, was traditionally implemented in GemStone with the position indicating an offset from 1; that is, the first position in the stream was 1.

ANSI specifies, and other Smalltalk dialects use, an offset of 0, so the first position in the stream is 0.

To allow legacy code and new code to coexist, GemStone includes sets of classes implementing both interfaces. There are four sets of classes, which all exist in the image (and therefore, may have instances), with only three sets being visible at any time. The following two sets are always visible:

- Legacy-style PositionableStream classes, compatible with previous GemStone version's PositionableStream classes:

```
PositionableStreamLegacy
   ReadStreamLegacy
   WriteStreamLegacy
```

- ANSI-compliant and portable PositionableStream classes:

```
PositionableStreamPortable
   ReadStreamPortable
   WriteStreamPortable
   ReadWriteStreamPortable
```

In addition, only one of the following sets is visible, depending on how your system is configured. These are two distinct sets of instances of Class, with the same name, but different implementations.

`PositionableStream` (with legacy definition and methods)

```
   ReadStream
   WriteStream
```

`PositionableStream` (with portable definition and methods)

```
   ReadStream
   WriteStream
```

The legacy versions are stored in Globals at: #GemStone_Legacy_Streams. The portable, ANSI-compatible versions are stored in Globals at: #GemStone_Portable_Streams.

To check what is currently installed, use the following methods:

```
PositionableStream class >> isLegacyImplementation
PositionableStream class >> isPortableImplementation
```

To install the portable version, use the method:

```
Stream class >> installPortableStreamImplementation
```

To install the legacy version, use the method:

```
Stream class >> installLegacyStreamImplementation
```

4.4 Sorting

You are likely at some point to want to present the contents of your Collection in a sorted order. There are a number of options, depending on the type of data you need to sort and the desired ordering.

- Some objects, such as Strings, Integers, and DateTimes, have an inherent sort ordering, and GemStone provides default sorts for Collections that contain only objects that can be compared using <=.
- *sortBlocks* allow you to specify expressions that can order any type of object according to your specific requirements.
- String data has special sorting requirements; for different applications, you may need sorting to consider case, accented characters, and language-specific sorting issues. When the simple default string sorting is not sufficient, specialized classes allow more control.

Default Sort

Objects representing numbers, dates and times, and strings have a natural sort order. If the collection contains objects that can be compared using <=, you can easily order the collection with the default sort.

Messages such as `sortAscending` and `sortDescending` can be sent to any collection that contain only these types of objects. For example:

Example 4.8

```
(Array with: 123 with: 3 with: 99 with: 10) sortDescending
%
 anArray( 123, 99, 10, 3)

(Array with: '123' with: '3' with: '99' with: '10')
   sortAscending
%
anArray( '10', '123', '3', '99')
```

It's more likely that you will want to sort more complex objects in your collection, such as Customers by name or Addresses by zip code. If the instance variables in your complex objects are objects that have a defined sort order, you can take advantage of `sortAscending:`, `sortDescending:`, and `sortWith:`, to provide a specification for the desired sort order.

For example, say we have a class for Employee, and define AllEmployees as a collection that contains instances of Employee:

Example 4.9

```
Object subclass: 'Employee'
   instVarNames: #( 'firstName' 'lastName' 'job' 'age')
   classVars: #()
   classInstVars: #()
   poolDictionaries: #()
   inDictionary: UserGlobals
%
Employee compileAccessingMethodsFor:
   #('firstName' 'lastName' 'job' 'age').
%

"Make some Employees and store them in a AllEmployees."
| Lee Kay Al myEmployees |
Lee := (Employee new) firstName: 'Lee'; lastName: 'Smith';
   job: #librarian; age: 40.
Kay := (Employee new) firstName: 'Kay'; lastName: 'Adams';
   job: #clerk; age: 24.
Al := (Employee new) firstName: 'Al'; lastName: 'Jones';
   job: #busdriver; age: 40.

myEmployees := IdentityBag new.
myEmployees add: Lee; add: Kay; add: Al.
UserGlobals at: #AllEmployees put: myEmployees.
%
```

To sort Employees by age and lastName, we can use the sortAscending: method, passing in the instance variables against which the ascending sort should be done:

Example 4.10

```
| returnArray tempString |
tempString := String new.
returnArray := AllEmployees sortAscending: #('age' 'lastName').
"Build a printable list of the sorted ages and lastNames"
returnArray do: [:i | tempString add: (i age asString);
         add: ' '; add: i lastName; lf].
tempString
%
24 Adams
40 Jones
40 Smith
```

For finer control, you can use the `sortWith:` method, which allows you to define direction for each instance variable

Example 4.11

```
| returnArray tempString |
tempString := String new.
returnArray := AllEmployees sortWith: #('age' 'Ascending'
                                         'lastName' 'Descending').
returnArray do: [:i | tempString add: (i age asString);
         add: ' '; add: i lastName; lf].
tempString
%
24 Adams
40 Smith
40 Jones
```

SortBlocks

You can also specify sort ordering by defining a sortBlock. A sortBlock is a two-argument block that should return true if the first argument should precede the second argument, and false if not. The expressions within the block are expected to by symmetrical - i.e., for two specific arguments for which the block returns true, then the block should return false when the arguments are reversed. If values compare equal, and the block returns the same results for both argument orders, then the final ordering of the equal elements is arbitrary.

SortedCollection is a type of Collection that includes a sortBlock; SortedCollection is discussed starting on page 56. You can use sortBlocks to sort the elements of any collection, using methods such as `sortWithBlock:`.

For example, to sort customers by last name:

```
AllEmployees sortWithBlock: [:a :b |
   a lastName <= b lastName]
```

You can create sort blocks that are as elaborate as you need; however, you should observe the symmetry of the expression.

For example, this block sorts by lastName, with further sorting by firstName if the lastNames are the same:

```
AllEmployees sortWithBlock: [:a :b |
   a lastName = b lastName
      ifTrue: [a firstName <= b firstName]
      ifFalse: [a lastName <= b lastName]
   ].
```

Sorting Large Collections

When sorting using the above methods, the entire collection must fit into memory. This may not be practical for very large collections. To avoid out of memory errors when sorting large collections, you can allow the sort to issue periodic commits, which will make the sort results persistent. Persistent objects don't need to stay in memory the way temporary objects do, which reduces the demand on memory.

These intermediate commits are enabled by specifying a persistentRoot for the sort, and by taking advantage of the IndexManager's ability to set up autoCommit. IndexManager is a class that manages Indexes, which you'll read more about in Chapter 7. You do not need to have an index on the collection in order to use this feature. However, you do need to set IndexManager's autoCommit setting to true. For more information on autoCommit, see page 111.

For example, the following code sorts AllEmployees collection using `sortWithBlock:persistentRoot:`

```
UserGlobals at: #SortedEmployees put: Array new.
System commitTransaction.
AllEmployees
   sortWithBlock: [:a :b | a lastName <= b lastName]
   persistentRoot: SortedEmployees
```

Chapter

5 String Classes and Collation

String handling is an important part of most applications. While Strings are a type of Collection, they have a number of unique features and behavior.

Characters and Unicode
Describes Characters.

CharacterCollection and String classes
Introduces the GemStone Smalltalk objects that store collections of Characters.

String Sorting and Collation
Describes collation, including traditional string collation and collation using the ICU libraries and Unicode strings.

Encrypting Strings
Explains how to encrypt strings.

5.1 Characters and Unicode

A Character is a special object -- an object whose value is encoded in the OOP. Literal Characters are formed with a leading $.

Code point

Each Character has a code or codePoint, which for lower order Characters is the ASCII value. Either of these terms may be used, though ASCII is an incorrect term for the higher code points. GemStone supports Characters with values from 0 to 16r10FFFF, the full Unicode range, except for the Unicode reserved range.

The Unicode range of codePoints from 16rD800-16rDFFF is reserved for encoding leading/trailing surrogate pairs for UTF-16 encoding. These can never be legal Unicode characters, and as such, it is an error to attempt to create a Character in this range.

To get the Character for a given codePoint, use the Character class methods `withValue:` or `codePoint:`.

Attributes

Characters have "type", and know if they are a digit, letter, separator, or other similar characteristic. This information is defined in the Unicode database as the Unicode general category, and a variety of testing methods are available. The Unicode database also defines the upper and lower case equivalents, and case conversion methods are available. See the image for a full list of available protocol.

For example,

```
$Z isUppercase
true

$u isDigit
false
```

Collation

Characters are ordered (collated) using internal character tables, which provide Unicode collation order for Characters up to code point 255. Characters above that are collated by code point. Character collation is used in collating instances of basic String classes. For more on collation, see "String Sorting and Collation" on page 75.

Character collation can be modified by installing character data tables, although this use is deprecated. This may be used to provide Unicode collation for Characters with codePoints above 255 or to provide legacy GemStone collate order (the collation order that was used in versions before 2.4). Character-based string collation has limitations outside the ASCII range; the ICU-library based string collation should be used if the default collation is not sufficient.

Unicode and the Unicode Database

The Unicode Consortium is an international standards organization that produces the Unicode Database. Unicode is a commonly used standard which provides unique codes for all Characters in all Character sets, in the range 0 to 0x10FFF. It also describes the category of each Character and relationship between it and other Characters, and provides a default collation order with the Default Unicode Collation Element Table (DUCET).

For more information on this database, refer to:

```
http://www.unicode.org/Public/UNIDATA/UCD.html
```

The Unicode Consortium provides code charts by script as well as a single master list of all characters, presented in an ASCII-only, comma-delimited version. The current version of this database can be found at

```
http://www.unicode.org/Public/UNIDATA/UnicodeData.txt
```

Character Data Tables

Customized Character data tables are deprecated. This information is provided for convenience when performing tests as part of transitioning to alternate character and string collation and other handling.

Character data tables are an internal structure that supports Character collation and Character-based string collation. For performance, the base installed tables include only Characters with codes 0..255. Installing character data tables allows Unicode collation of

Characters over 255, or collation in GemStone legacy collate order, which avoids the need for existing applications to rebuild indexes.

The character data tables are used repository-wide, and changing this table may have consequences; if the character tables change collation, then GemStone indexes and collections that depend on ordering may not return the correct results.

Installing Character Data Tables

The following methods can only be executed by SystemUser. After commit, the installed tables apply to all logins for the repository. Any indexes or collections that depend on string collation must be rebuilt after installing new tables. Installed tables are not affected by upgrade or conversion. The tables distributed with GemStone are based on Unicode version 5.1; note that this is an older version of the Unicode standard.

- To install a 0...255 character table in GemStone legacy collate order:

```
Character installCharTables: (PassiveObject fromServerTextFile:
   '$GEMSONE/examples/CharTableDefault.tab') activate.
```

- To install the full Unicode character table in GemStone legacy collate order:

```
Character activateCharTablesFromFile:
   '$GEMSTONE/examples/CharTableUnicode510.dat'.
```

- To install the full Unicode character data table in Unicode DUCET collate order:

```
Character activateCharTablesFromFile:
   '$GEMSTONE/examples/CharTableUnicode410.dat'
```

- To reset to the default internal character table:

```
Globals removeKey: #CharacterDataTables.
```

- To disable loading of installed character tables on session login, set the following environment variable:

```
export GS_DISABLE_CHARACTER_TABLE_LOAD=TRUE
```

5.2 CharacterCollection and String classes

CharacterCollection and String classes

CharacterCollection is a subclass of SequenceableCollection that is specialized to hold Characters, and expands the protocol inherited from SequenceableCollection to include messages specialized for comparing, searching, concatenating, and changing the case of character sequences. CharacterCollection is the abstract superclass for strings, including String class and other specialized strings. In this discussion, we will generally use the term String to include all the subclasses of CharacterCollection, not just the String class itself.

Each element of a CharacterCollection is a Character. A Character has an associated value, which may require more than one byte of physical storage. This is handled for you by GemStone; if more storage is required, the String is transparently converted to the appropriate type. For String, this is DoubleByteString or QuadByteString; for Unicode7, this is Unicode16 or Unicode32. The specific class does not change the interaction with the

object; access by index will return the Character at the given index, regardless of how many bytes the Character actually requires. However, if you need to write the String to a file or other non-GemStone sequential format, this may require converting to an appropriate single-byte format, generally UTF-8.

The CharacterCollection hierarchy includes the following concrete classes:

Strings

These classes are traditional strings.

String
Strings hold Characters with codepoints in the range 0..255.

DoubleByteString
DoubleByteStrings are required when one or more Characters in a string needs more than one byte of storage. DoubleByteStrings holds Characters with codepoints in the range 0...16rFFFF (64K).

QuadByteString
QuadByteStrings are required when one or more Characters in a string needs more than two bytes of storage. QuadByteStrings holds Characters with codepoints in the range 0...16r10FFFF.

Unicode Strings

For strings that require locale-specific collation, specialized subclasses of the String classes are provided. These classes rely on the open-source ICU libraries to provide comparison and sorting behavior.

Unicode7
A subclass of String, limited to holding Characters with codepoints in the range 0..127 that are represented in 7 bits.

Unicode16
A subclass of DoubleByteString, holding Characters with codepoints in the range 0...16rFFFF (64K), excluding the range 16rD800-16rDFFF. This range is reserved for surrogates that allow encoding into UTF-16.

Unicode32
A subclass of QuadByteString, holding Characters with codepoints in the range 0..16r10FFFF. Again, this excludes the range range 16rD800-16rDFFF.

Symbol

Class Symbol is a subclass of String. Each Symbol with a unique set of Characters is guaranteed to have only one canonical instance in GemStone. Symbols are created by a special process, the SymbolGem, to ensure this uniqueness.

Like Strings, symbols may also contain Characters with values that require more than a byte of storage, and will convert into DoubleByteSymbols or QuadByteSymbols as needed. GemStone Smalltalk uses symbols internally to represent variable names and selectors. All symbols may be viewed by all users. Private information should be maintained in Strings, not in Symbols.

Symbols, DoubleByteSymbols, and QuadByteSymbols are restricted to 1024 or fewer characters.

You can "create" symbols using `asSymbol` or `withAll:` method. If the Symbol was created previously as part of the GemStone kernel, by another user, or by yourself, you will get the existing Symbol. A new symbol is only created if it has not been previously defined. Existing Symbols cannot be modified.

Since Symbols are canonical, the class of a Symbol always depends on the contents. While you can create a DoubleByteString with only characters in the range of String, you cannot create a DoubleByteSymbol that does not contain at least one character in the DoubleByte range, and the same is true for QuadByteString.

Symbols that have no references from anywhere in the system will eventually be garbage collected, if the system is configured to do so. See the *System Administration Guide* for more information on symbol garbage collection.

ByteArray

ByteArray is a specialized collection that is restricted to holding Integers between 0 and 255 (inclusive).

Instances of ByteArray can be creating using literal syntax #[]. For example:

```
#[ 1 2 3 4 ]
```

Utf8

Utf8 is a subclass of ByteArray. It holds UTF-8 encoded bytes created by sending `encodeAsUTF8` to a string, or by reading encoded data from a GsFile using `contentsAsUTF8`. Utf8 instance should not be directly created or edited.

```
'šamas' encodeAsUTF8
anUtf8( 197, 161, 97, 109, 97, 115)
```

Instances of Utf8 can be decoded into a Unicode String using the method `asUnicodeString`. This will create an instance of Unicode7, Unicode16 or Unicode32 string, whatever is the smallest representation that can hold the decoded characters.

Instances of Utf8 can be read from and written to instance of GsFile, which cannot directly handle characters with codePoints over 256.

String equality, ordering, and interoperation

By default, traditional strings and symbols are compared for equality and ordered using legacy rules. String and symbol are ordered using a character based comparison,and equality includes non-printing characters as well as printing characters. The collation is described in more detail on page 75.

Unicode strings always use a an IcuCollator, and comparison is based on the entire string, and is highly configurable; the character data tables are not used. Unicode string equality does not consider non-printing characters. Unicode collation is described starting on page 76.

Since legacy and Unicode equality and ordering rules are different, traditional strings and symbols, using legacy comparision, cannot be ordered with Unicode strings (other than using protocol that explicitly provides a collator). To avoid inconsistent results, attempting to do so results in an error.

When Unicode Comparison Mode is enabled (see page 81), traditional strings and symbols are also collated using Unicode rules, and can be ordered and compared with Unicode strings in collections.

String protocol

Creating Strings

You have already seen strings created as literals. Strings created as literals are invariant; they cannot be modified after they are created.

In addition to creating strings literally, you can use the inherited instance creation methods, such as `new:` and `withAll:`. For example:

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

To create a string that can be modified later, you can use `withAll:` with a literal String:

```
| myString |
myString := String withAll: 'azure'.
myString at: 1 put: $A.
myString
Azure
```

Concatenating Strings

A string responds to the comma operator by returning a new string in which the argument to the comma has been appended to the string's original contents. For example:

```
'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, GemStone Smalltalk creates a String object for the first literal, `'String'`. The #, message returns a new instance of String containing `'String con'`, and the second #, message creates 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), which modifies the original string. Note that you cannot start with a literal string, since a literal string is invariant.

For example:

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

Converting Strings

CharacterCollection and its subclasses define messages that let you perform various conversions.

Strings can be converted into other kinds of strings.

- Instances of traditional strings can be converted to the lowest-storage type of Unicode string using `asUnicodeString`.
- Instances of Unicode strings can be converted to the lowest-storage type of traditional strings using `asString`.
- Either kind of String can be converted to the lowest-storage type of Symbol using `asSymbol`.

For example:

```
'abcde' asSymbol
#'abcde'
```

Strings can be converted in case:

- `asUppercase` creates a new instance with all uppercase letters
- `asLowercase` creates a new instance with all lowercase letters
- `asTitlecase` creates a new instance with the first letter of each word capitalized, the remaining letters lowercase.
- `asFoldcase` returns a new instance in "fold case", which is case-free for comparison and usually is similar to the lowercase.

For example:

```
'abcde' asUppercase
ABCDE
```

Strings can be converted to numbers and other types of objects as well. Note that not all Strings can be converted to all kinds of other objects —if the String does not contain the representation of a number, for example, it's meaningless to convert it to an Integer, so this will return an error.

For example:

```
'15' asFloat
1.5000000000000000E01
```

Equality and Identity

Traditional strings are equal to each other if they contain the exact same Characters in the same case; equality is case-senstitive.

Unicode strings compared using = follow the ICU library comparsion rules for equality, which are similar, although any non-whitespace control characters (such as null) are ignored for the comparison.

Traditional strings and Unicode strings cannot be compared to each other for equality using =. Any comparison involving a Unicode string require a collator. To compare traditional and Unicode strings in any combination, use `compareTo:collator:`. Specifying nil for the collator uses the default collator.

Strings can be compared for case-insensitive equality using the methods `isEquivalent:` or `equalsNoCase:`.

Identity in Literal vs. nonliteral

Literal and nonliteral 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.

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

However, literal strings that contain the same character sequences and are compiled at the same time are both equal and identical:

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

This distinction can become significant in building sets. If you add both *litString1* and *litString2* to the same IdentitySet, the set will contain only one instance of `'abc'`; however, an IdentitySet would include both *nonlitString1* and *nonlitString2*.

Searching and Pattern matching

CharacterCollection and its subclasses define methods that can tell you whether a string contains a particular sequence of characters and, if so, where the sequence begins. This search can be case sensitive, case insensitive, and may include wild cards.

Below are some common methods; see the image for further methods.

Case-sensitive Search	Case-insensitive Search	Description
	includesString: *subString*	Return true if the receiver includes *subString*.
findString: *subString* startingAt: *anIndex*	findStringNoCase: *subString* startingAt: *anIndex*	Return the index of *subString* if it exists within the receiver at *anIndex* or above, otherwise zero (0).
matchPattern: *patternArray*		Return true if the receiver matches the specifications in *patternArray*
findPattern: *patternArray* startingAt: *anIndex*	findPatternNoCase: *patternArray* startingAt: *anIndex*	Return the index of a substring in the receiver that matches the specifications in *patternArray* at *anIndex* or above, otherwise zero (0).

Pattern Matching Wild Cards

Pattern matching arguments (*patternArray*) consist of an Array containing combinations of Strings and the wildcard characters `$*` and `$?`. The character `$?` matches any single character in the receiver, and `$*` matches any sequence of characters in the receiver.

This is an example of the use of wildcard characters in pattern matching.

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

Since $* is interpreted as "any sequence of characters", this returns true.

Similarly, The following example returns the index at which a sequence of characters beginning and ending with $r occurs in the receiver.

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

If a wildcard character `$*` or `$?` occurs in the receiver or within a string in the argument array, it is interpreted literally.

The following expressions illustrate what happens when the `*` is within the string and interpreted literally:

```
'w*r' matchPattern: #('weimaraner')
false

'weimaraner' findPattern: #('w*r') startingAt: 1
0
```

5.3 String Sorting and Collation

While strings clearly have a natural sort order, the details of that order are complex. Different languages may sort the same set of strings differently, according to the particular rules in that language. Even within one language, different applications may want to order string data differently. To complicate matters, some languages may treat certain sequences of characters as a unit when sorting strings.

The sorting of strings into a standard order is called collation. Collation depends on the results of a comparison between two strings, which in turn depends on how the Characters within the string are collated. While this simple view breaks down with some sorting requirements and linguistic rules, basic string comparison is adequate for many uses and is faster than the more complete external collation.

Traditional String Legacy Collation

Traditional strings (String, DoubleByteString, and QuadByteString) and symbols (Symbol, DoubleByteSymbol, and QuadByteSymbol) are collated by individual character. The comparison of characters with values up to 255 are done according to the Default Unicode Collation Element Table (DUCET), and Character 256 and above are sorted by codePoint, the Unicode numeric value.

This is the default behavior for traditional strings and symbols. Installing non-default Character Data Tables (see "Character Data Tables" on page 68) will affect the Character collation, according to the specific table installed. Enabling Unicode Comparison Mode

(see "Unicode Comparison Mode" on page 81) causes traditional strings and symbols to collate following the same rules as Unicode strings. This section does not apply when using Unicode Comparison Mode.

String ordering using <= (as well as <, >, and >=) is not case-sensitive. When instances of String, DoubleByteString, and QuadByteString are compared using <= or related operations, the comparison first is done case-insensitive. If they are found to be equal other than with respect to case—if the only difference is case—then they are collated according to the Character Data Table, which specifies uppercase comes before lowercase.

For example:

```
#( 'c' 'MM' 'Mm' 'mb' 'mM' 'mm' 'x' )
   sortAscending
anArray( 'c' 'mb' 'MM' 'Mm' 'mM' 'mm' 'x' )
```

Since ordering is by character, with only case being excluded, the default ordering is sensitive to accents and other diacritical marks on characters. Characters with diacritical marks are not related to the base character.

For example, all words beginning with 'Co' and 'co' would sort before all words beginning with 'Có' and 'có':

```
#('Cór' 'COz' 'Coa' 'cóa')
   sortAscending
anArray( 'Coa', 'COz', 'cóa', 'Cór')
```

Unicode String Collation using ICU libraries

The classes IcuLocale and IcuCollator provide an interface to the ICU (International Components for Unicode) libraries. The ICU libraries are a widely-used, open-source implementation of language-specific sorting and collation.

For a complete explanation of the features and subtleties of language-specific collation, you should refer to documentation on the ICU website.

```
http://icu-project.org/
```

Unicode strings (instance of Unicode7, Unicode16, and Unicode32) and instances of Utf8 use IcuCollator and IcuLocale to perform sorting operations using the ICU libraries. The collation is performed by considering the entire string, not on a character-by-character basis, and requires a specific language and locale to determine the rules for the comparison. In addition to specific language rules, ICU sorting is highly configurable for other application-specific sorting requirements.

While collation will vary according to specific language and locale, in general ICU collation orders characters with diacritical marks with the base character, and sorts lowercase before uppercase.

For example, using the sorting examples in the previous section and the default collator for the US, a different sort ordering is produced from that of legacy collation:

```
( 'c', 'mb', 'mm', 'mM', 'Mm', 'MM', 'x')

( 'Coa', 'cóa', 'Cór', 'COz')
```

By configuring the IcuCollator, however, other orderings, including ordering similar to the traditional string collation, may be produced..

Only Unicode strings and Utf8 instances use ICU sorting, by default. You may explicitly order traditional strings and symbols by specifying an IcuCollator; and enabling Unicode Comparison Mode (page 81) will cause all these to use ICU comparison and sorting.

IcuLocale

Instances of IcuLocale represent a specific language, country, and language variant. The available IcuLocales are in the shared library and can be listed using `IcuLocale class >> availableLocales.`

A default instance of IcuLocale is instantiated on first reference, and stored in session state. The default IcuLocale is based on the operating system locale setting for the gem.

To set a specific default IcuLocale, use the method `IcuLocale class > default:`. This sets the default locale for the session executing this code. While the instance of IcuLocale can be made persistent, the default IcuLocale does not persist from session to session.

`IcuLocale class >> getUS` is an example of instantiating an IcuLocale for the language English in the country US.

To determine what IcuLocale is currently in use, use the method `IcuLocale >> default`.

Example 5.1 Default IcuLocale

```
topaz 1> printit
IcuLocale default
%
a IcuLocale
  name                en_US
```

IcuCollator

An IcuCollator encapsulates the rules involved in collation for a specific IcuLocale. A default instance of IcuCollator is instantiated on first reference, based on the default IcuLocale, and stored in session state.

When comparing instances of Unicode String classes, the comparison always uses an IcuCollator, using the method `compareTo:collator:`. If an IcuCollator is not specified, such as when Unicode String classes are compared using >, the `IcuCollator default` is used; which in turn uses `IcuLocale default`.

You can also create an instance of IcuCollator for a specific locale, if you need to use a specific collation rules other than the default. You can do this using IcuCollator class methods `forLocale:` *anIcuLocale* or `forLocaleNamed:` *aString*. For example, to create an IcuCollator for the German language as spoken in Germany:

```
IcuCollator forLocaleNamed: 'de_DE'
```

The actual string comparison is done by the ICU libraries, and follows the ICU comparison rules for that locale. Collation rules are similar in most western languages, but there are differences in specific languages.

For example, in the Hungarian language, 'cs' is considered a single letter, so words that start with 'cs' are sorted together and follow other words beginning with 'c'. The following example sets up a collection that is sorted according to Hungarian rules:

Example 5.2 Sorting in Hungarian IcuLocale

```
| hungarianWords collator |
collator := IcuCollator forLocaleNamed: 'hu_HU'.
hungarianWords := IcuSortedCollection newUsingCollator:
   collator.
hungarianWords
   add: 'csak' asUnicodeString;
   add: 'cukor' asUnicodeString;
   add: 'comb' asUnicodeString.
hungarianWords
a IcuSortedCollection
  sortBlock           a ExecBlock2
  collator            a IcuCollator
  #1 comb
  #2 cukor
  #3 csak
```

Customizing Sort

IcuCollator includes a number of attributes that can be used to customize the sort. These attributes work within the specific language rules of the associated IcuLocale.

Keep in mind that while the default values and the descriptions listed in Table 1 apply to most locales, particularly with non-Western scripts, the defaults may be different in different locales, and the attribute may have different behaviors.

See the ICU site, particularly the pages under:

```
http://userguide.icu-project.org/collation
```

for more precise descriptions and more detailed documentation.

Table 1 IcuCollator Attributes

Attribute name	Allowed values	Default	
alternateHandling	true | false	false	When true, allows space and punctuation characters within the string to be ignored.
caseFirst	'off', 'upperFirst', or 'lowerFirst'	'off'	When comparing case, determines if upper or lowercase is sorted first. Most locales sort lowercase first when caseFirst is 'off' as well as when 'lowerFirst'.
caseLevel	true | false	false	When true, considers case in the comparison, even if the strength would normally not consider case. For some locales, adds another strength level between SECONDARY and TERTIARY strengths.
frenchCollation	true | false	false	When true, sorts secondary differences (the same base character with different diacritical marks) in reverse order (starting from the end of the string). This is the correct collation in French.
normalization	true | false	false	Determines whether to normalize input strings, useful if input data may be un-normalized. Has performance impact.
numericCollation	true | false	false	When true, sorts numeric sequences within the string by numerical rather than string comparison; e.g. sort '100' after '2'.
strength	PRIMARY - 0 SECONDARY - 1 TERTIARY - 2 QUARTENARY - 4, or IDENTICAL - 15	TERTIARY	Determines the level of collation factors to consider, such as diacritical marks and case. See discussion below for more details.

Strength allows degrees of sort, to consider or not consider things like accent characters and case when performing the sort. The default strength is TERTIARY for most locales (the main exception being Japanese). The following are the sort strengths:

- PRIMARY sorts by primary differences, ignoring secondary and later differences. The base letter represents a primary difference, so for example 'a' and 'b'.
- SECONDARY sorts by primary and secondary differences, ignoring tertiary and later differences. An example of a secondary difference is diacritical differences on the same base letter, for example 'o' and 'ó'.

- TERTIARY sorts by primary, then secondary, then tertiary differences. Uppercase vs. lowercase is a tertiary differences. TERTIARY is the default sort order for most locales.
- QUATERNARY is used in Japanese, where it distinguishes between Japanese Katakana and Hiragana, and can be used to break ties among separator characters when alternateHandling is true.
- IDENTICAL sorts by the specific character, by codepoints in the NFD (Normalization Form Canonical Decomposition) form. There is a performance impact with this strength.

The default sort strength is TERTIARY. As an example, when two strings are compared using TERTIARY strength, characters in the strings are compared first by the base character, ignoring any case or diacritical marks. If the base characters are the same, they are compared by diacritical mark, ignoring case. If both base characters and diacritical marks are the same, then case is considered. Note that unlike GemStone's Strings or ASCII ordering, the default sorts places lowercase before uppercase.

Keep in mind that with lower sort strengths, when a factor such as case is not used, the relative position in the results of similar strings is not deterministic; the strings compare as the same, and so their position will depend on the order of the input.

By using the IcuCollator sort attributes, you have a great deal of control over your specific sorting.

For example, using the alternative handling example, you can sort strings that include spaces, dashes and other punctuation without considering the punctuation characters when doing the comparison:

Example 5.1 Sort ignoring punctuation

```
| blues collator|
collator := IcuCollator forLocale: IcuLocale default.
collator alternateHandling: true.
blues := IcuSortedCollection newUsingCollator: collator.
blues add: (Unicode7 withAll: 'blue berry').
blues add: (Unicode7 withAll: 'blue moon').
blues add: (Unicode7 withAll: 'bluebird').
blues add: (Unicode7 withAll: 'blue bird').
blues add: (Unicode7 withAll: 'blue-bird').
blues add: (Unicode7 withAll: 'bluetooth').
blues
%
a IcuSortedCollection
  sortBlock            a ExecBlock2
  collator             a IcuCollator
  #1 blue berry
  #2 bluebird
  #3 blue bird
  #4 blue-bird
  #5 blue moon
  #6 bluetooth
```

IcuSortedCollection

An IcuSortedCollection is a specialized subclass of SortedCollection for which you do not set the sortBlock. An IcuSortedCollection is associated with a IcuCollator, which is turn is associated with an IcuLocale, and the sorting behavior is specific to the configuration of these instances. IcuSortedCollections rely on the open-source ICU libraries to perform the comparisons and produce correctly collated results.

Using IcuCollator is recommended if you will have sorted collections containing Unicode strings. This avoids lookup failures if a different collator is used to lookup than was used to sort the elements in the collection.

Unicode Comparison Mode

Configuring your repository to use Unicode Comparison Mode allows traditional strings to automatically use Unicode comparison rules. This permits traditional and Unicode strings to be compared interchangably.

Unicode Comparison Mode affects not only collation using >, >=, <, <=, but also equality using = and ~=. The legacy and Unicode rules for equality are not identical.

Since Unicode Comparison Mode affects equality comparisons of traditional strings and symbols, as well as ordering, it may break lookup in existing hashed collections in addition to SortedCollections and indexes. The risks and impacts of Unicode Comparison Mode are unquantified, and there is no support for application validation after changing comparison mode; working with GemTalk Engineering is recommended, and careful testing, if you wish to experiment with Unicode Comparison Mode

Unicode Comparison Mode is controlled by the Global #StringConfiguration. By default, StringConfiguration is set to String, which provides legacy string comparison mode.

To turn on Unicode mode, as SystemUser, execute

```
Globals at: #StringConfiguration put: Unicode16.
System commitTransaction.
```

The current session is not affected; the new mode will take effect for all subsequent logins.

To disable Unicode Comparison Mode, as SystemUser, execute

```
Globals at: #StringConfiguration put: String
System commitTransaction.
```

The current session is not affected; the new mode will take effect for all subsequent logins.

5.4 Encrypting Strings

There are times when you may which to encrypt strings in your repository or for transmittal to other systems. GemStone provides an interface to Advanced Encryption Standard (AES) encryption/decryption, provided by the OpenSSL open source libraries included with GemStone.

The AES specification is available at:

```
http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf
```

All encryptions/decryptions are in cipher block chaining (CBC) mode; see the AES specification document for further details.

Encryption and decryption API methods are provided for 128-bit/16-byte keys, 192-bit/24-byte keys, and 256-bit/32-byte keys, using the following methods.

Encryption can be done on instances of ByteArray or Uft8, or subclasses of CharacterCollection. For encryption, you must provide a key that is a ByteArray of the appropriate size (16, 24, or 32 bytes) containing key bytes, and a salt that is a 16-byte ByteArray containing salt values.

The following methods encrypt or decrypt using the specified key and salt, return the encrypted or decrypted result:

```
aesEncryptWith128BitKey: aKey salt: aSalt
aesDecryptWith128BitKey: aKey salt: aSalt

aesEncryptWith192BitKey: aKey salt: aSalt
aesDecryptWith192BitKey: aKey salt: aSalt

aesEncryptWith256BitKey: aKey salt: aSalt
aesDecryptWith256BitKey: aKey salt: aSalt
```

These methods place the encrypted or decrypted result into *aByteObjOrNil*, starting at offset 1, and resizing if necessary. If *aByteObjOrNil* is nil, a new instance of the same class as the receiver will be created containing the results.

```
aesEncryptWith128BitKey: aKey salt: aSalt into: aByteObjOrNil
aesDecryptWith128BitKey: aKey salt: aSalt into: aByteObjOrNil

aesEncryptWith192BitKey: aKey salt: aSalt into: aByteObjOrNil
aesDecryptWith192BitKey: aKey salt: aSalt into: aByteObjOrNil

aesEncryptWith256BitKey: aKey salt: aSalt into: aByteObjOrNil
aesDecryptWith256BitKey: aKey salt: aSalt into: aByteObjOrNil
```

You may use `ByteArray withRandomBytes:` *N* to produce pseudo-random key and salt values for encryption.

For example:

```
topaz 1> run
| key salt encrypted |
key  := ByteArray withRandomBytes: 32.
salt := ByteArray withRandomBytes: 16.
encrypted := 'My secret string' aesEncryptWith256BitKey: key
   salt: salt.
encrypted aesDecryptWith256BitKey: key salt: salt.
%
My secret string
```

Chapter

6 Numeric Classes

This chapter describes GemStone's numeric classes. These include Integers, floating point (limited-precision rational numbers), fractions (arbitrary precision rational numbers), and decimal numbers. Most numbers can be specified as literals within your code, and most numbers can be used in expressions with, or converted to, other types of numbers.

Integers
Describes classes that represent whole numbers: SmallInteger and LargeInteger.

Binary Floating Point
Describes classes for binary floating point numbers: SmallDouble and Float.

Other Rational Numbers
Describes classes for other rational numbers with different ranges and precisions, including Fraction, FixedPoint, ScaledDecimal, and DecimalFloat.

Internationalizing Decimal Points using Locale
How to control the display of decimal points.

Random Number Generator
Information on the set of random number generator classes, providing random numbers of various purposes.

6.1 Integers

Integers in GemStone are composed of SmallIntegers and LargeIntegers. Most Integers you are likely to use will be SmallIntegers, in the range of -2^{60} to $2^{60}-1$. Integers outside this range are represented by LargeIntegers. Operations that result in a value outside the SmallInteger range transparently result in LargeIntegers, and vice-versa

The literal syntax for Integer will create either a SmallInteger or LargeInteger.

SmallInteger

SmallIntegers are special (immediate) objects, that is, the number itself is encoded in the OOP, making instances of this class both small (since no further storage is required) and

fast. They are also unique, so SmallIntegers of the same value are always identical (==) as well as equal (=).

SmallIntegers have a range from -2^{60} to 2^{60} -1. Values outside this range must be represented as LargeIntegers.

LargeInteger

LargeIntegers are not special objects; they require an OOP.

Each instance of LargeInteger is stored as an array of bytes, where every 4 bytes represents a base 4294967296 digit. The first 4 bytes are the sign digit (0 or 1), the next 4 bytes in that array constitute the least significant base 4294967296 digit, and the last 4 bytes are the most significant base 4294967296 digit.

Instances of LargeInteger have a maximum size of 4067 digits plus the sign. The maximum absolute value for a LargeInteger is $(2^{130144} - 1)$. Attempting to create a LargeInteger that exceeds this maximum will fail with an Integer overflow error.

Printing Integers

Integers are printed by default, using `Integer>>asString`, in base 10. You may print using other bases by invoking `printStringRadix:` or `printStringRadix:showRadix:`.

For example,

```
1234 printStringRadix: 2
'10011010010'

-1234 printStringRadix: 16 showRadix: true
'-16r4D2'
```

6.2 Binary Floating Point

Floating point values in GemStone are composed of SmallDoubles and Floats. The most commonly used floating points will be SmallDoubles. While both SmallDouble and Float represents 8-byte binary floating point numbers, as defined in IEEE standard 754, SmallDoubles have a reduced exponent range. Some floating point values therefore can only be represented by instances of Float, rather than SmallDouble. Similarly to SmallInteger and LargeInteger, GemStone operations return one or the other as needed.

The numerical behavior of instances of Float is implemented by the mathematics package of the vendor of the machine on which the Gem process is running. There are slight variations in results with different platform's implementation of the IEEE-754 standard.

You can get the components of a floating point value using the methods `signBit`, `exponent`, and `mantissa`.

SmallDouble

SmallDoubles are special objects; as with SmallIntegers, the number itself is encoded in the OOP, making instances small and fast. They are also unique, so SmallDoubles of the same value are identical (==) as well as equal (=).

Each SmallDouble contains a 61 bit value, in IEEE format but with reduced exponent range. There is 1 sign bit, 8 bits of exponent and 52 bits of fraction. SmallDoubles are always in big-endian format (both on disk and in memory).

SmallDoubles can represent C doubles that have value zero or that have exponent bits in range 0x381 to 0x3ff, which corresponds to about 5.0E-39 to 6.0E38; approximately the range of C 4-byte floats.

Float

Floats are not special objects; they require an OOP.

Each Float contains a 64 bit value in IEEE format, with 1 sign bit, 11 bits of exponent and 52 bits of mantissa. Floats are in cpu-native byte order when in memory, and the byte order of the extent when on disk.

In addition to the finite numbers, the IEEE standard defines floating point formats to include Infinity (positive and negative) and NaNs (not a Number), which can be quiet or signaling. NaNs results from an operations whose result is not a real number, such as:

```
-23 sqrt
%
PlusQuietNaN
```

Infinity results from operations that return a value outside the range of representation, such as:

```
32.0 / 0
%
PlusInfinity
```

ExceptionalFloats are named, unique instances of Float, not of SmallDouble. Exceptional Floats include:

PlusInfinity
MinusInfinity
PlusQuietNaN
MinusQuietNaN
PlusSignalingNaN
MinusSignalingNaN

Since the sign of NaNs is not defined, GemStone operations return only positive NaNs; they do not return MinusQuietNan or MinusSignalingNan.

An unusual quality of NaNs is that they are not equal to themselves. This means that NaNs can cause problems if used as keys of hashed equality-based collections.

```
PlusQuietNaN = PlusQuietNaN
%
false
```

Literal Floats

Literal numbers in evaluated code that include a decimal point by default create a SmallDouble or Float. If the value is in the SmallDouble range, a SmallDouble will be created, otherwise a Float will be created.

Literal floats may be specified using exponential notation. For example, 5.1E3 and 5.1E-3 are valid SmallDouble literals.

Note that using a plus sign before the exponent is not allowed in literal floats, although it can be used to create floating points from strings (using `Float fromString:`). This avoids ambiguity with Smalltalk dialects that would interpret this as the addition operator. For example, 5.1E+3, which historically GemStone would interpret as the same as 5.1E3, is disallowed; code must either omit the +, or include white space to clarify the addition operator.

Printing Binary Floating Points

SmallDoubles and Floats are printed by default, using `asString`, in exponential notation. For readability, you can format floating point numbers for printing with `asStringUsingFormat:`.

This method accepts an Array of three elements:

- an Integer between -1000 and 1000, specifying a minimum number of Characters in the result String. Negative arguments pad with blanks to the left, positive arguments pad to the right. Note that if the value of this element is not large enough to completely represent the Float, a longer String will be generated.
- an Integer between 0 and 1000, specifying the number of digits to display to the right of the decimal point. If the printed representation of the float requires fewer characters, the result is padded with blanks on the right. If the value is insufficient to completely specify the float, the value is rounded to fit.
- A Boolean indicating whether or not to display the magnitude using exponential notation. If true, exponential notation is used; if false, decimal notation.

For example:

```
12.3456 asStringUsingFormat: #(-8 2 false)
'   12.35'

12.3456 asStringUsingFormat: #(4 10 true)
'1.2345600000E01'
```

By default, Float and SmallDouble are printed with the equivalent of #(1 16 true).

6.3 Other Rational Numbers

For some application, binary floating points are problematic, since there are common decimal values that cannot be expressed exactly in binary floating point; for example, 5.1 does not have a precise binary floating point representation.

```
5.1
%
5.0999999999999996E00
```

There are several options to avoid this: Fraction, FixedPoint, ScaledDecimal, and DecimalFloat. These classes are independent of each other, and each provides different qualities of precision and range.

Fraction

Fractions precisely represent rational numbers. Fractions are composed of an integer numerator and an integer denominator. As the ratio of two Integers, Fractions can represent any rational number to an unbounded level of precision.

The display of fractions is as the numerator and denominator separated by the $/$ character, which is also the division binary method. Fractions have no literal representation. An expression such as 1/3, which performs a division of two Integers, will return a Fraction if the result is not an Integer.

```
(1/3) printString
%
1/3
```

Any expression, not just division expressions, that could result in Fractions will be reduced automatically, to the lowest Fraction or to an Integer.

```
(5/6) + (1/6)
%
1
```

FixedPoint

FixedPoints, like Fractions, represents rational numbers, but also include information on how they should be displayed. A FixedPoint is composed of an integer numerator, integer denominator, and an integer scale. Like Fraction, this allows rational numbers to be represented with unbounded precision, and since fractional arithmetic is used in calculations, numerical results do not lose precision.

The scale provides automatic rounding when representing the FixedPoint as a String.

FixedPoint uses a literal notation using p, such as 1.23p2.

ScaledDecimal

ScaledDecimals represent a decimal number to the precision of a fixed number of fractional digits. ScaledDecimals are composed of an integer mantissa and a power-of-10 scale. While ScaledDecimals represent decimal fractions to the precision specified, not all values can be represented exactly by ScaledDecimals. The maximum scale is 30000.

ScaledDecimals use a literal notation using s, such as 1.23s2.

ANSI does not precisely specify the scale of a ScaledDecimal that is returned by mathematical operations. The following rules are used:

- For unary messages, the scale of the result equals the scale of the receiver.
- For a one-argument message, the scale of the result is the greater of the scale of the receiver and argument. An integer receiver or argument coerced to a ScaledDecimal should effectively have a scale of zero, meaning the result will have the scale of the non-coerced ScaledDecimal argument or receiver.

For some mathematical operations, the returned value type is a ScaledDecimal, but the returned value cannot always be exactly represented as a ScaledDecimal with the correct scale. In these cases, the results are rounded using the following rules:

- Following the example of IEEE754 float rounding, the ScaledDecimal that is answered is selected as though we computed the numerically exact value and then chose the closest representable ScaledDecimal of the scale specified by the rules. If the numerically exact value falls exactly halfway between two adjacent representable ScaledDecimal values of the scale specified by the rules, the ScaledDecimal with an even least significant digit is answered.

DecimalFloat

DecimalFloats represent base 10 floating point numbers, as defined in IEEE standard 854-1987.

Objects of class DecimalFloat have 20 digits of precision, with an exponent in the range -15000 to +15000. The first byte has encoded in it the sign and kind of the floating-point number. Bit 0 is the sign bit (0=positive, 1=negative). The values in bits 1 through 3 indicate the kind of DecimalFloat:

```
001x = normal
010x = subnormal
011x = infinity
100x = zero
101x = quiet NaN
110x = signaling NaN
```

Bytes 2 and 3 encode the exponent as a biased 16-bit number (byte 2 is more significant). The actual exponent is calculated by subtracting 15000. Bytes 4 through 13 form the mantissa of the number. Each byte holds two BCD digits, with bits 4 through 7 of byte 4 containing the most significant digit.

Similarly to Float, operations that would not result in a real number, or that produce a result outside the representable range, result in Exceptional numbers:

DecimalPlusInfinity
DecimalMinusInfinity
DecimalPlusQuietNaN
DecimalMinusQuietNaN
DecimalPlusSignalingNaN
DecimalMinusSignalingNaN

6.4 Internationalizing Decimal Points using Locale

The class Locale allows you to obtain operating system locale information and use or override it in GemStone. GemStone currently only uses the decimalPoint setting, to provide localized reading and writing of numbers involving decimal points. Updates to Locale are stored in session state, and only persist for the lifetime of the session. They are not affected by commit or abort.

Note that Smalltalk syntax requires the use of "." as the decimal point separator, so expressions involving literal floating point numbers within Smalltalk code will still require use of the period, regardless of Locale.

To override the operating system locale information, use the following message:

`Locale class >> setCategory:` *categorySymbol* `locale:` *LocaleString*

The *LocaleString* passed to `setCategory:locale:` must be defined on the host machine. You can use the UNIX command **locale -a** to get a list of all available LocaleStrings.

While there are a number of Locale category symbols, the only ones that are of use in this release are `#LC_NUMERIC` and `#LC_ALL`, either of which will set the category that affects the decimal point.

For example, To use decimal localization appropriate for Germany:

```
Locale setCategory: #LC_NUMERIC locale: 'de_DE'.
```

To reset to UNIX default value, using period:

```
Locale setCategory: #LC_ALL locale: 'C'.
```

The following method returns the decimalPoint setting for the current Locale:

```
Locale decimalPoint
```

Regardless of Locale, you can read a string with a period decimal point using the methods:

```
Float class >> fromStringLocaleC:
DecimalFloat class >> fromStringLocaleC:
```

6.5 Random Number Generator

The class Random and its subclasses provide random number generation.

There are two types of random number generation, which correspond to separate subclass hierarchies. The SeededRandom subclasses provide random numbers generated within GemStone code, using a starting seed value. The HostRandom subclass provides access to the host operating system's `/dev/urandom` random number generator. This is much slower, but unlike the SeededRandom numbers, on some platforms this may be designed to be cryptographically secure.

The class hierarchy of the Random classes are:

```
Object
   Random (abstract)
      HostRandom
      SeededRandom (abstract)
         Lag1MwcRandom
         Lag25000CmwcRandom
```

Random

The Random class is an abstract superclass for the random number generators. It also can be used to create an instance of the default random number generator class, Lag25000CmwcRandom. Sending instance creation messages to Random will return instances of Lag25000CmwcRandom. For example:

```
Random seed: 12345
%
aLag25000CmwcRandom
```

The message

```
Random new
```

will create an instance of Lag25000CmwcRandom that is seeded with numbers generated from the host OS `/dev/urandom`.

Once you have an instance of a concrete subclass of Random, you can generate random numbers or collections of random numbers with the following range and type specifications:

`float` - a random Float in the range [0,1)

`floats:` *n* - a collection of *n* random floats in the range [0,1)

`integer` - a random non-negative 32-bit integer, in the range [0,2^{32}-1]

`integers:` *n* - a collection of *n* random non-negative integers in the range [0,2^{32}-1]

`integerBetween:` *l* `and:` *h* - a random integer in the range [*l*,*h*]. *l* and *h* should be less than approximately 2^{31}.

`integers:` *n* `between:` *l* `and:` *h* - a collection of *n* random integers in the range [*l*,*h*]. *l* and *h* should be less than approximately 2^{31}.

`smallInteger` - Answer a random integer in the SmallInteger range, [-2^{60},2^{60}-1]

Subsequent calls to the same instance will generate new random numbers.

You should create an instance of a Random subclass and retain that to generate many random numbers, rather than creating new instances of a Random subclass.

HostRandom

HostRandom allows access to the host operating system's `/dev/urandom` random number generator.

HostRandom is much slower than the other subclasses of Random. However, `/dev/urandom` on some platforms may be intended to be cryptographically secure random number generator, which none of the other subclasses are. It also has the advantage of not needing an initial seed, and so is good for generating random seeds for the faster Random subclasses.

HostRandom uses a shared singleton instance, which is accessed by sending #new to the class HostRandom. Sending #new has the side effect of opening the underlying file `/dev/urandom`. This file normally remains open for the life of the session, but if you wish to close it you can send #close to the instance, and later send #open to reopen it. If you store a persistent reference to the singleton instance the underlying file will not be open in a new session and you must send #open to the instance before asking for a random number.

Since HostRandom is a service from the operating system, it cannot be seeded, and should not be used when a repeatable random sequence of numbers is needed.

SeededRandom

SeededRandom is an abstract superclass for classes that generate sequences of random numbers that can be generated repeatedly by giving the same initial seed to the generator.

In addition to creating new instances using the class methods new and seed:, the following instance methods allow repeatable sequences to be generated:

`seed:` *aSmallInteger*
: Sets the seed of the receiver from the given seed, which can be any SmallInteger. The subsequent random number sequence generated will be the same as if this generator had been created with this seed.

`fullState`, `fullState:` *stateArray*
: The internal state of a generator is more than can be represented by a single SmallInteger. These messages allow you to retrieve the full state of a generator at any time, and to restore that state later. The random number sequence generated after the restoration of the state will be the same as that generated after the retrieval of the state. You might, for instance, allow a generator to get its initial state from `/dev/urandom`, then save this state so the random sequence can be repeated later.

Lag1MwcRandom

Lag1MwcRandom is intended for internal use only, in creating a random seed for instances of Lag25000CmwcRandom. Lag1MwcRandom is slower, is not perfectly fair, and has a shorter period, so the only advantage is its ability to be seeded by a single 61-bit SmallInteger, rather than a seed of more than 800000 bits as required by Lag25000CmwcRandom.

Lag25000CmwcRandom

Lag25000CmwcRandom is a seedable random generator with a period of over 10^{240833}. It is a lag-25000 generator using the complementary multiply-with-carry algorithm to generate random numbers. Its period is so long that every possible sequence of 24994 successive 32-bit integers appears somewhere in its output, making it suitable for generating random n-tuples where n<24994. Its output is fair in that the number of 0 bits and 1 bits in the full sequence are equal.

While this generator is recommended for most uses, it is **not** cryptographically secure, so for applications such as key generation you should consider using HostRandom, once you satisfy yourself that HostRandom is secure enough on your operating system.

You can also allow the seed bits to be initialized from the HostRandom, then retrieve that state by sending #fullState. That state can later be restored by sending the retrieved state as an argument to #fullState:.

Chapter 7

Indexes and Querying

This chapter describes GemStone Smalltalk's indexing and querying mechanism, a system for efficiently retrieving elements of large collections.

Overview
reviews the concept of relations.

Defining and Executing Queries
describes the structure of query predicates, the types of queries, and how to construct and execute a query.

Creating Indexes
discusses GemStone Smalltalk's facilities for creating indexes on collections.

Special Kinds of Queries and Indexes
Describes how to create indexes and query on Unicode Strings, enumerated and collection-valued path terms, and other special cases.

Managing Indexes
How to perform index management: find out about indexes in your system, remove existing indexes, handle errors, and audit indexes.

Query Formulas and Optimization
How to use Query formulas, and how these formulas are optimized.

7.1 Overview

Most applications require one or more databases : a set of data that is critical for business function, and needs to be accessed efficiently. In GemStone, this is a represented as a instance of collection that holds instances of business objects. For large applications, collection may be very large, containing many thousands or even millions of objects; and you will need to be able to find specific objects within that collection quickly and easily.

The following example shows employee data in table form:

Figure 7.1 Employees

Name	Job	Age	Address
Fred	clerk	40	22313 Main, Dexter, OR
Sophie	busdriver	24	540 E. Sixth, Renton, WA
Conan	librarian	40	999 Walnut, Hilt, CA

In GemStone Smalltalk, you would naturally define Employee and Address classes that are subclasses of Object, with *name*, *job*, *age*, and so on as instance variables; and create instances of these classes to store your employee information. You'd put these instances in a collection. While various kinds of collections could be used, an IdentitySet (or IdentityBag) would be the logical choice, since the Employees are not inherently ordered.

To make it easy to associate behavior with your set of Employees, you might define a class SetOfEmployees that is a subclass of IdentitySet. Then, you might make the collection of Employee globally accessible, by referencing it by the key #myEmployees.

Recall that in UnorderedCollections, lookup is by value, rather than by position in the collection or by a key. Standard Collection protocol allows you to locate an object in an UnorderedCollection by select:, or similar messages.

For example:

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

Searching this way sends one or more messages to each element of the receiver. In this example, the messages age and = are each sent once for each element of myEmployees. This is fine for small collections, but becomes unreasonably slow for collections containing many thousands of complex objects. This is particularly true if objects are not in memory and need to be read from disk in order to respond to messages.

GemStone Indexes

Indexes provide a way to locate specific objects in a collection by value. Indexing a Collection creates internal structures such as balanced trees (Btrees). Only a few message sends are needed to lookup a value or range of values in the indexing structures. When collections are indexed, they can return results without having to iterate the collection or send messages to each object.

Indexes may only be created and indexed queries performed over collection classes that are subclasses of UnorderedCollection: this includes IdentityBag, IdentitySet, Bag, Set, RcIdentityBag, and a few other specialized kinds of Set.

Indexes are created on objects based on instance variables, not on message sends; since the instance variable relationships are known by the system, indexes can be usually be updated automatically as elements are added and removed from the collection, and when object instance variables change value. There are some exceptions to this, but these require manual updating.

What you need to do

In order to take advantage of efficient indexed queries on your collection, you need to do the following:

- Formulate the query that you wish to be indexed, using query syntax
- Create an index on that collection, that specifies that particular instance variable path on which you will perform the query.
- execute the query on that indexed collection, using specific query protocol

For example, if you want searching for Employee by name to be fast, you can create an index on name, and then use query syntax to execute a query on name. If you need to search based on different instance variables, such as searching by ID as well as by name, you need to create multiple indexes.

You may use query syntax to create and execute a query, even when there is no index on that variable, but it will have to iterate the collection to find the results.

The details of how to define and create indexes, and how to formulate and execute queries, are described in following sections.

APIs for creating indexes and queries

There are two distinct APIs for performing these tasks.

- ❒ The GsIndexSpec/GsQuery interface is introduced in version 3.2, and provides ways to define and manipulate indexes and queries using objects.
- ❒ The traditional indexing API, which creates indexes using UnorderedCollection methods and performs queries using selection blocks, is an alternate way to use indexes. Some features are not available using this traditional indexing API. Differences are noted in the following sections.

Internal index structures and behaviors are the same for both APIs, and if indexing and querying is limited to the common features, they can be used interchangeably.

Managing Indexes

In addition to creating indexes and queries, you will also need to do some management on your indexes and queries. For example, you should evaluate your indexes for performance, remove indexes that are no longer needed, and audit indexes to ensure the structures are correct. Many of these indexing tasks are handled by IndexManager.

Indexing trade-offs

Creating an index creates a number of internal objects that implement the indexing infrastructure; this includes the btree structure and the RcIndexDictionary. If your collection is small, the extra overhead of this infrastructure reduces the value of the indexed search. The size at which the trade-off makes an index worthwhile will depend on many factors, and testing is always preferred. As a rule of thumb, if your collection contains fewer than about 2000 objects, it may not be worthwhile to create an index.

Building and removing indexes requires extra management. You must be sure to remove indexes before dereferencing instances of UnorderedCollection, to avoid leaving indexing structures in place that cannot be automatically garbage collected.

Special Syntax for Indexing

GemStone indexing uses several syntactical elements that are either specific to, or primarily used for, index creation and indexed queries.

Path-dot syntax

Indexes are created, and queries formed, using special syntactic structure called a *path*, which designates variables for indexing and describes certain features of the index. Path syntax uses a period . to represent the object/instance variable name relationship.

For example, for a collection of Employees, in which each employee has an address instance variable, which refers to an Address that has a "state" instance variable, an example of a path is:

```
address.state
```

In the simplest case, a path on an instance variable on the collection elements, this is just the instance variable name. For example:

```
name
```

Each instance variable name on the path is a *pathTerm*. In the above example, `address` and `state` are each pathTerms. Paths can contain many pathTerms, if the elements of the collection represent a deeply nested tree of objects. If each object has the appropriate instance variable, this is an example of a longer path:

```
account.order.address.state
```

You may specify the values of variables nested up to 16 levels deep within the elements of a collection.

Path-dot syntax can be used anywhere in GemStone code; it is required in index creation and queries, for which message sends are not allowed.

An initial `'each.'`, where each represents the elements of the collection, is recommended but optional for GsIndexSpec index creation. For example:

```
each.address.state
```

This "`each`" is not permitted in paths supplied to UnorderedCollection index creation methods.

Enumerated pathTerms

A vertical bar | in the path indicates the presence of two alternate instance variables that will be indexed together, as if they were a single variable.

For example, you might want to search on both name and nickname in a single operation. This might look like this:

```
account.name|nickname
```

This syntax can be used to create indexes and queries using the GsQuery/GsIndexSpec API, but not with the traditional API.

Set-value path terms

An asterisk * in the path indicates a collection, which must be an instance of an indexable class (an instance of a subclass of UnorderedCollection).

For example, if the instance variable children contains an IdentityBag of instances of Child, and a child has the instance variable age:

```
children.*.age
```

This syntax can be used to create indexes and queries using the GsQuery/GsIndexSpec API, but not with the traditional API.

7.2 Defining and Executing Queries

Before you can define indexes on your collection, you need to determine the ways in which you will need to search your collection to retrieve elements. Once you have defined the queries you need, this will determine the details of the indexes you need to create.

At its simplest, a query consists of an instance variable of the objects in the collection, a comparison operator, and a literal to which the value is compared. For example, if you wish to be able to find all employees 18 and older, your query formula would be something like this:

```
each.age >= 18
```

In this example, every object in the collection (`each`) has an instance variable `age`, which is specified using dot-path notation. The value of that instance variable is compared, greater than or equal, to the literal 18.

This formula is simple; you can formulate queries based on multiple instance variable values, operators, and constants, and combine them using boolean logic. However, note that in the query syntax, you cannot include message sends. The details of predicate syntax is described in the next section, 'Query Predicate Syntax'.

Equality vs. Identity queries

Queries, and the indexes that support them, can be based on equality comparisons or on identity comparison. Equality comparisons include greater than and less than comparisons, as well as equal and not equal; equality comparison-based indexes depend on being able to order the indexed elements respective to one another, so an element or range of elements can be found using search on the internal btree.

Identity indexes and queries compare elements by identity and have more limited flexibility, but fewer restrictions—since any objects can be compared using identity. You cannot query for a range of results using Identity indexes.

Query Predicate Syntax

A query contains a *predicate* expression, which is a Boolean expression that, when evaluated with the elements of the collection, returns true or false. In a query, the expression usually compares an instance variable on the collection objects with another instance variable or with a constant.

A predicate contains one or more *predicate 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. For example, in

```
each.age >= 18
```

`each.age` and `18` are operands, while >= is a comparison operator. The only time you would not have a comparison operator is if the operand is itself a Boolean (true or false).

Predicate Operands

An operand can be a path (*each.age*, in this case), a variable name, or a literal (18, in this example). All GemStone Smalltalk literals except arrays are acceptable as operands.

Predicate Operators

The following tables list the comparison operators used in a query predicates:

Table 1 Comparison Operators for Identity Indexed Queries

==	Identity comparison operator
~~	Non-identity comparison operator

Table 2 Comparison Operators for Equality Indexed Queries

=	Equality comparison operator
~=	Not-equal comparison operator
<	Less than equality operator
<=	Less than or equal to equality operator
>	Greater than equality operator,
>=	Greater than or equal to equality operator

No other operators are permitted in a GsQuery or selection block query.

These operators behave according to the rules governing the objects being compared. Equality comparisons that use an index, however, are more restricted in an indexed query; you cannot compare classes that are not in the same hierarchy as the class specified during index creation. The exception to this are strings; all kinds of traditional strings can be compared to each other and to Symbols, and all kinds of Unicode strings can be compared to each other, but not to traditional strings or Symbols. (But see the note on page 113).

Nil is a special case for equality index comparisons. Because of its special significance as a placeholder for unknown or inapplicable values, nil is comparable to every kind of object, and every kind of object is comparable to nil. Since the appearance of nil signifies a value that is not there, less than and greater than comparison results will not include nil values.

Basic Classes optimized for indexes

The most efficient indexes are on certain GemStone Smalltalk kernel classes, in which all or a large part of the object's value can be encoded within the index internal structures, avoiding the need to read the object itself. These classes are referred to as "Basic classes" for indexing and querying.

This includes the following classes, and subclasses of these classes:

Character, String, DoubleByteString, QuadByteString,
Unicode7, Unicode16, Unicode32,
Symbol, DoubleByteSymbol, QuadByteSymbol,
Boolean, Time, Date, DateTime, DateAndTime
SmallInteger, LargeInteger,
ScaledDecimal, Fraction,
SmallDouble, Float, DecimalFloat

Instances of basic classes, and subclasses of basic classes, use specialized protocol to perform the comparisons.

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

Creating indexes on Unicode strings (Unicode7, Unicode16, and Unicode32) require using a Unicode index.

Using non-basic Classes

You can create indexes on instances of classes that are not basic classes, including classes you defined yourself.

Identity indexes on instances of your own classes, since they compare on the identity of the objects, require no addition consideration, nor do indexes on instances of classes that inherit a kernel implementation of the equality operators.

If you need to, you can redefine the equality operators `=`, `~=`, `<=`, `<`, `>=`, and `>` in classes that you have created that are not subclasses of Basic classes. There are some caveats; this is discussed under "Redefined Comparison Messages" on page 116.

Combining Predicates using Boolean Logic

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) or disjunction (logical OR) operator.

The conjunction operator, `&`, makes the predicate true if and only if the terms it connects are true. The disjunction operator, `|`, makes the predicate true if either one, or both, of the terms it connects are true.

Selection block queries only permit the conjunction operator; GsQuery queries allow both conjunction and disjunction operators.

You may also negate individual predicate terms using `not`, only in GsQuery queries.

Each predicate term must be parenthesized.

For example, the following are legal queries. This example returns a collection of employees who are named Conan and are librarians:

```
(each.name = 'Conan') & (each.job = 'librarian')
```

While this returns a collection of employees who are 40 or younger in age, or who are not librarians.

```
(each.age <= 40) | (each.job = 'librarian') not
```

Combining Range Predicates

Queries that use less than or greater than, such as each.age >= 18, define a starting (or ending) point in a range query. Specifying both a starting point and ending point creates a range query. For example,

```
(18 <= each.age) & (each.age <= 65)
```

Using the GsQuery API, but not in the traditional API, these two terms can be combined into single range predicate. For example:

```
18 <= each.age <= 65
```

Range specifications such this can only be defined with this syntax if the operands and comparison operators truly define a range.

Selection Block Queries

To create a query based on your search criteria, you can use either GsQuery or traditional selection block syntax. Both types of query syntax produce the same results. Your existing GemStone indexes will use the selection block syntax; GsQuery has the advantage of allowing programmatic management of the query, and there are a number of specialized query features that are only available when using GsQuery.

Selection Blocks

Selection blocks are a kind of block specialized for queries, using curly braces instead of brackets. The compiler understands this syntax and creates the selection block instance when the code or method is compiled. Selection blocks require exactly one argument and do not allow message sends within them; there are other restrictions on allowed operations within a selection block, as described under "Query Predicate Syntax" on page 97.

A selection block query might be written like this:

```
{:each | each.address.state = 'OR'}
```

As with the equivalent iteration methods that use ordinary blocks with square brackets, `each` represents the elements of the collection, and in this case, the dot-path syntax resolves the object at the instance variable #address, which object has an instance variable #state.

In selection block queries, you can reference temporary, instance or other variables within the block, and these are resolved at runtime as in ordinary blocks.

Executing Selection Block Queries

A selection block is used with `reject:`, `collect:`, `detect:` and `detect:ifNone:`, to perform the query over a collection.

For example:

```
Employees select: {:each | each.address.state = 'OR'}
```

Selection blocks can be also used with `reject:`, `collect:`, `detect:` and `detect:ifNone:`. These have the same semantics as with standard blocks executed on a collection. For example, `reject:` will return a result set that includes all elements for which the block evaluation would return false.

Return values

A selection block query returns a new instance of collection of the same class as the base collection. So if you query on a collection that is an instance of SetOfEmployees, for example, the results will be returned in an instance of SetOfEmployee.

The exception is if you root collection is an instances of a reduced-conflict (Rc) collection, such as RcIdentityBag. The result of a query in this case is a non-Rc collection. The results of a query on a RcIdentityBag are returned as an instance of IdentityBag. This avoids the overhead that supports the reduced-conflict classes.

The collection returned from a query has no index structures. If you want to perform indexed selections on the new collection, you must build the necessary indexes on the new collection.

Queries using GsQuery

GsQuery is a programmatic way to define a query, allowing you to easily abstract, store and reuse various aspects of the query. While GsQuery provides more query features, the query is internally similar to the processes used by selection block queries, and the query results will be the same.

Creating and Executing a GsQuery

To create a query using GsQuery, you create an instance of GsQuery, specifying the details of the search. There are a number of ways to specify the search; the most simple is by passing in a string. For example:

```
GsQuery fromString: 'each.age >= 18'
```

This message will return an instance of GsQuery. Before it can be executed, it must be bound to a collection. You can create the GsQuery using `fromString:on:` in order to create a GsQuery that is bound to a particular collection, or you can bind the collection later using the `on:` method. Sending `queryResult` to the GsQuery will return the results of the query.

The following two examples illustrate creating and executing a bound query, and creating and executing a query that is not associated with a collection, and binding it before execution.

```
(GsQuery fromString: 'each.age >= 18' on: Employees)
   queryResult.

(GsQuery fromString: 'each.age >= 18')
   on: Employees;
   queryResult.
```

Since the `fromString:` protocol requires a string, if the query includes literal strings, they must be double quoted. For example:

```
GsQuery fromString: 'each.firstName = ''Fred'''.
```

Creating a GsQuery from a selection block

If you have existing code that includes selection block queries, you can use those selection blocks to create the instances of GsQuery.

For example,

```
GsQuery fromSelectBlock: {:each | each.address.state = 'OR'}
```

This can be bound using `on:`, or created using `fromSelectBlock:on:`, similarly to how you create and bind a GsQuery from a string.

You may also create the GsQuery from a saved query formula, previously extracted from another GsQuery; this is described on page 124.

Query Variables

The strings used to define GsQuery instances may contain variables—any element of a predicate that is are not a literal or path-dot expressions. This allows your query to be stored and executed later using different values.

For example, for a query such as

```
GsQuery fromString: '18 <= each.age <= 65'
```

This can be generalized to a query with variables:

```
GsQuery fromString: 'min <= each.age <= max'.
```

The resulting formula in the GsQuery includes 'min' and 'max' as variables. These must be bound to specific values before the query can be executed. Binding is done by sending the `bind:to:` message to the query. For the above example, to execute the query:

```
aQuery := GsQuery fromString: 'min <= each.age < max'.
aQuery
   bind: 'min' to: 18;
   bind: 'max' to: 65;
   on: myEmployees;
   queryResult
```

Note that the “max” and “min” in the query formula are string elements, and are not affected by any temporary or instance variables named max or min in the scope of the code being executed. The only way to resolve max and min are by binding variables.

GsQuery’s Collection protocol

To get the query results, you can send `queryResult` to the instance of GsQuery. GsQuery accepts other Collection protocol, which it responds to as if the GsQuery were the query result Collection.

You can send `asArray` or `asIdentityBag` to the GsQuery directly, for example:

```
(GsQuery fromString: 'each.address.state = ''OR'''
   on: Employees) asArray
```

Performing one of the collection operations that are provided for GsQuery simplifies your code, since you may not have to put results in temporary variables. It may or may not allow you to avoid creating query result objects. Enumeration methods also allows you to perform code while the query is executing, rather than waiting for the results.

GsQuery and cacheQueryResults

By default, each time you execute any GsQuery collection protocol, the query is performed again. So sending `includes:` or `isEmpty` to a GsQuery, by default, does not avoid executing the query during a subsequent `queryResult`.

You can cache the results of your GsQuery by specifying an instance of GsQueryOptions with `cacheQueryResult:` set to true. This will cache the result set of the GsQuery. Note that this cache will not reflect changes in the root collection that occurred after the query was executed; you are responsible for re-running the query if current results are required.

You may pass in an instance of GsQueryOptions using the `fromString:options:` and related protocol for GsQuery.

For example:

```
query := (GsQuery fromString: 'each.address.state = ''OR'''
   options: (GsQueryOptions cacheQueryResult)
   on: Employees).
query isEmpty ifTrue: [^'no results'].
report := self createReportingStructure.
query do: [:ea | report updateDataWith: ea].
...
```

GsQuery enumeration methods accepting blocks

Among the collection protocol that GsQuery understands are the methods `do:`, `select:`, `reject:`, `collect:`, `detect:` and `detect:ifNone:`. While these look similar to fetching query results using selection blocks, since the actual query is already provided by the GsQuery, there are key differences.

With GsQuery, it's important to remember that these will operate on the *result set* of the initial query. In essence, you are adding an additional, non-indexed search criteria to the indexed query. This additional code will be executed for each element in the collection for which the indexed query matches, at the time that the index query is examining that result element.

For example, if you have an index on Employee age, and a query such as:

```
(GsQuery fromString: 'each.age <= 18' on: Employees)
```

Using this query, you can add an additional search criteria using `select:`, so that only Employees who live in Oregon are returned.

```
(GsQuery fromString: 'each.age <= 18' on: Employees) select:
   [:each | each state = 'OR']
```

This will return a result set that includes Employees under 18 who live in Oregon. The `state` message is only sent to the elements (Employees) who are under 18, it is not executed for every element in the collection.

Order of results

Provided there is an index on the query path, the enumeration block operates on each object in the result set in the order specified by the index. However, since the `select:` or other method will necessarily return a kind of UnorderedCollection (see "Return values"

on page 104), the objects in the collection returned by the enumeration method will be not be ordered.

You can use the enumeration protocol to produce results that are ordered according to the index. For example:

```
resultArray := Array new.
(GsQuery fromString: 'each.age <= 18' on: Employees) do:
   [:each | resultArray add: each].
```

However, for ordered results, you may want to stream over the results instead.

Efficiency of query vs. enumeration

It is more efficient to perform an indexed query using GsQuery than to add additional criteria using enumeration methods.

For example, the following code returns a collection of all employees who are 26 or younger, and who respond false to `hasOtherHealthInsurance`.

```
GsQuery fromString: 'each.age <= 26' on: myEmployees)
   reject: [:each | each hasOtherHealthInsurance]
```

This is useful if you have predicates that require message sends. However, if you can formulate the second statement as an indexable predicate, it would be more efficient. If `hasOtherHealthInsurance` was actually an instance variable, you could write this as:

```
(GsQuery fromString: '(each.age <= 26) &
   (each.hasOtherHealthInsurance) not' on: myEmployees)
   queryResults
```

Early exit from execution

Since the code in the block provided to `select:` (and similar methods) is executed for each element that the indexed query itself would return, this provides a way to exit the indexed query early. In this block, you can execute any code (as long as it does not modify the collection or the objects in the collection in ways that would change the result set). Based on this code, if it's no longer useful to continue the search, you can exit the block.

For example, say you have a collection of purchase orders, and you are generating a report of all open purchase orders. If a new order arrives during the period you are executing this operation, you might want not want to bother producing the already-obsolete report.

```
(GsQuery fromString: 'each.isOpen' on: MyOrders) do:
   [:anOrder |
   report add: anOrder description.
   self checkForNewOrders ifTrue: [^'report cenceled']
   ]
```

Return values

`GsQuery >> queryResult` will, like selection block queries, return a new instance of collection of the same class as the base collection, unless protocol such as `asArray` are used to specify the class of the results.

Also similarly to selection block queries, queries on instances of reduced-conflict (Rc) collections, return the equivalent non-Rc collection.

The collection returned from a query has no index structures. Indexes belong to specific instances of collections, rather than the classes. If you want to perform indexed selections on the new collection, you must build the necessary indexes on the new collection.

Query results as Streams

It may be more useful to return the result of an equality query as a stream, instead of a collection, especially if the result set is large. Returning the result as a stream not only is faster, is also avoids the need to have all the result objects in memory simultaneously.

Streaming on index results return the results in order that is defined by the index, so you can iterate over the elements that are returned in the order defined by the index, with no extra effort.

To get the results as a stream, use the message `GsQuery >> readStream`. or `UnorderedCollection >> selectAsStream:`. These methods return an instance of RangeIndexReadStream, which is similar to a ReadStream but specialized for index results.

You can then iterate the results using standard stream protocol. Instances of RangeIndexReadStream understand the messages `next`, `atEnd`, and similar ReadStream protocol.

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. You should not modify the objects in the base collection while streaming, nor add or remove objects; doing so can cause an error or corrupt the stream.

For example, suppose your company wishes to send a congratulatory letter to anyone who has worked there for thirty 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*, and there is an index on this, you can use a stream to formulate the query as follows:

```
oldTimers := (GsQuery fromString: 'each.lengthOfService >= 30'
   on: myEmployees) readStream.
[ oldTimers atEnd ] whileFalse: [
   | anEmployee |
   anEmployee := oldTimers next.
   anEmployee sendCongratuations. ].
```

The selection block query interface uses the message `selectAsStream:` to create a stream on the query results. This is handled the same as a GsQuery readStream.

```
oldTimers := myEmployees selectAsStream:
   {:anEmp | anEmp.lengthOfService >= 30}.
[ oldTimers atEnd ] whileFalse: [
   | anEmployee |
   anEmployee := oldTimers next.
   anEmployee sendCongratuations. ].
```

Limitations on streamable queries

Streams on query results have certain limitations; for example, the predicate in the query must be logically streamable. The following restrictions apply:

- It takes a single predicate only; no conjunction of predicate terms is allowed. The exception is with if two predicates can be automatically combined to form a single range predicate. So, for example, `(each.age > 18) & (each.age <= 65)` is legal, since it can be reformulated as a single range predicate, `(18 < each.age <= 65)`.
- The predicate can contain only one path.
- The collection you are searching must have an equality index on the path specified in the predicate.

7.3 Creating Indexes

To execute a query efficiently, you need to create an index on the instance variables that you want to query on. These indexes provide a mapping from the specific key values that you are interested in to the results (the objects in the collection).

The path you provide when creating an index provides the key. These keys are objects in the collection, or the values of a specific instance variable within the elements of a collection.

For example, given a collection of Employees, and the path `each.address.state`, the objects at the `state` instance variable (perhaps a two-character String) would be the key. Then when you make an indexed query for Employees with addresses in a given state, that `state` key is used to lookup the matching elements (instance of Employee).

Equality and Identity Indexes

Indexes fall into two main types: **Equality Indexes** and **Identity Indexes**. Equality indexes keep keys in a btree structure, which provides tree-based lookup. This allows greater-than and less-than, and sorted range results, to be produced. IdentityIndexes store keys in a hashed identity dictionary, which only allows lookup by identity.

When creating an index, you specify whether an equality or identity index is created.

Then, in the indexed query, the comparison operator controls the type of index that is used. Queries containing >, >=, <, <=, =, and ~= use an equality index. Queries containing == and ~~ will look for an identity index.

If you only have an identity index on a variable, but form your query using an equality operator, the query will not have an index to use (and thus, will iterate the collection).

The exception to this is if your equality index is on a "special" object, such as a SmallInteger, SmallDouble, Character, or Boolean, in which equality and identity are the same. This results in an implicit index (see page 108), which can be used to make identity based queries.

You may create both equality and identity indexes on the same path.

Specialized subtypes of Indexes

Within the general types of indexes, there are some variations with special features.

Unicode Indexes

The **Unicode Index** is a type of Equality Index that allows you to index instances of Unicode strings— Unicode7, Unicode16, and Unicode32—which require a IcuCollator to compare. See "Unicode String Indexes and Queries" on page 113 for details.

Reduced-conflict Equality Indexes

An **Rc Equality Index** is a type of Equality Index in which internal indexing structures are reduced-conflict. This avoids some transaction conflicts when creating an index on a reduced-conflict (RC) collection, such as RcIdentityBag. Reduced-conflict classes are described in "Indexes and Concurrency Control" on page 137. Rc Equality indexes are described under "Reduced-Conflict Indexes" on page 110.

Implicit Indexes

Most of the indexes you will use for your queries are created explicitly, by executing code specifying a particular indexed path.

However, under some cases, implicit indexes are also created as a side effect. These indexes can be used to perform indexed queries. You cannot manage or remove them, however; they can only be removed by removing the primarily explicit index for which the implicit index is a side-effect.

Implicit indexes include:

- In the process of creating an index on a nested instance variable, GemStone Smalltalk also creates identity indexes on the values that lie on the path to that variable.

  For example, if you create an equality index on `'name.lastName'`, it also creates an identity index on name. By creating this index, you can make indexed identity queries on the objects specified by name, without explicitly creating an index on name.

- An implicit identity index is also present if you create an equality index on a Special, such as a SmallInteger, SmallDouble, Character, or Boolean, in which equality and identity are the same.

Creating indexes using GsIndexSpec

To create an index using GsIndexSpec, do the following:

1. **Create an instance of GsIndexSpec**

   This is done by executing `GsIndexSpec new`

2. **Define one or more indexes on the Spec**

   To define an index, send an index creation message to the GsIndexSpec, including the path you want indexed, the class of the last element (for equality indexes), and options (if used).

   Index creation messages include, for example, `equalityIndex:lastElement-Class:` and `identityIndex:` (see the list below).

3. **Create the index on a specific collection**

   To actually create the index, send the message `createIndexesOn:`, providing the specific collection on which you want to create the indexes.

To put this all together, for example:

```
GsIndexSpec new
   identityIndex: 'each.userId';
   equalityIndex: 'each.age' lastElementClass: SmallInteger;
   equalityIndex: 'each.address.state' lastElementClass: String;
   createIndexesOn: myEmployees.
```

This creates an identity index on userId, an equality index on age, and another equality index on address.state, all on the collection `myEmployees`.

You can view the indexes by recreating the specification code, using `indexSpec`. For example:

```
myEmployees indexSpec
GsIndexSpec new
   identityIndex: 'each.userId';
   equalityIndex: 'each.age'
      lastElementClass: SmallInteger;
   equalityIndex: 'each.address.state'
      lastElementClass: String;
   yourself.
```

The expressions that create a GsIndexSpec can be stored as instances or as code, and can be used along or in conjunction with other GsIndexSpec instances, to create the same set of indexes or a customized set of indexes on any collections that contain objects that implement the given paths.

The following index creation methods are defined on GsIndexSpec :

```
equalityIndex:lastElementClass:
equalityIndex:lastElementClass:options:
identityIndex:
identityIndex:options:
unicodeIndex:
unicodeIndex:collator:
unicodeIndex:collator:options:
```

GsIndexOptions

A instance of GsIndexOptions specifies optional additional refinements that will be used when creating a particular index on a collection. A GsIndexOptions instance is provided by using the variants of the index creation methods that include the options: keyword. GsIndexOptions are not used by traditional index creation.

The options available for GsIndexOptions are:

`GsIndexOptions class >> reducedConflict`
: Returns an instance of GsIndexOptions that specifies that the index is reduced-conflict. This applies for equality indexes, making these into Rc Equality Indexes.

`GsIndexOptions class >> optionalPathTerms`
: Returns an instance of GsIndexOptions that specifies that the collection is allowed to be non-homogenous, that each element of the collection is not required to include all indexed instance variables on the path. May applies to any index.

GsIndexOptions can be combined using the plus operator and removed using the minus operator. For example:

```
GsIndexSpec new
   equalityIndex: 'each.name'
   lastElementClass: String
   options: (GsIndexOptions optionalPathTerms +
      GsIndexOptions reducedConflict)
```

Creating indexes using UnorderedCollection protocol

UnorderedCollection provides protocol to create indexes. This creates the same index structures as GsIndexSpec, but does not provide access to some index features.

The following index creation methods are defined on UnorderedCollection:

```
createIdentityIndexOn:
createEqualityIndexOn:withLastElementClass:
```

For example, this message requests that myEmployees creates an identity index on the instance variable *userId* within each of its elements:

```
myEmployees createIdentityIndexOn: 'userId'.
```

And these messages request to create equality indexes on the instance variables *age* and *name*:

```
myEmployees
   createEqualityIndexOn: 'age'
   withLastElementClass: SmallInteger.
myEmployees
   createEqualityIndexOn: 'address.state'
   withLastElementClass: String.
```

Together, these three statements create the same indexes that were provided in the example in the previous section.

Reduced-Conflict Indexes

When creating an equality index on a collection that is reduced-conflict, such as RcIdentityBag, some multi-user commit conflicts may be avoided by creating the indexing structures themselves as reduced-conflict. This option is not particularly useful if your collection is not reduced conflict (such as IdentitySet, etc.), since this collection will encounter any commit conflicts as well.

This doesn't apply to identity indexes, which are always reduced-conflict.

If you are creating an index on an RcIdentityBag, to make the index reduced-conflict, use an index creation method with the `options:` keyword, and pass in `GsIndexOptions reducedConflict`. For example:

```
GsIndexSpec new
   equalityIndex: 'each.name'
   options: (GsIndexOptions reducedConflict)
```

Using UnorderedCollection index creation protocol to create an index, the message is:

```
UnorderedCollection >> createRcEqualityIndexOn:withLastElementClass:
```

Optional pathTerms

A homogenous collection is one in which each element in the indexed collection defines the instance variable described by the index, for each pathTerm in the indexed path. By default, indexes require that the collection be homogeneous. If any element does not have the given instance variable, it will raise an error when the element is added to the collection.

If you want to create an index on a non-homogenous collection, you can define the indexes with optional pathTerms using GsIndexSpec protocol. Use an index creation method with the `options:` keyword, and pass in `GsIndexOptions optionalPathTerms`. For example:

```
GsIndexSpec new
   equalityIndex: 'each.nickName'
   options: (GsIndexOptions optionalPathTerms)
```

When creating an optional pathTerm index, it is not an error when the objects in the collection do not implement an instance variable specified by the index. For a multi-pathTerm index, that includes each pathTerm; objects with missing instance variable definitions for any of the pathTerms in the indexed path are not considered when creating query results.

If you create an index with a pathTerm for an instance variable that does not exist at all (perhaps due to a typing error), then the index is created correctly and does not report an error, even if it does not create the index you might have intended to create.

While Indexes are Being Created

Indexing a large collection will take some amount of time to create the infrastructure and tracking for each indexed object.

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

Queries during index creation

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 an object that is actively participating in the index at the same time, index creation is terminated with an error.

Auto-commit

Creating or removing an index creates and/or modifies many objects related to the internal structures that support indexes. These modifications are uncommitted changes that must be kept in the session's memory until these changes are committed. Many uncommitted changes place a large demand on memory and creates a risk of out of memory conditions. Chapter 8, "Transactions and Concurrency Control," explains uncommitted objects and transactions in more detail, while Chapter NN explains object memory use.

To avoid problems during index creation, it is often necessary to set the IndexManager to autoCommit. The IndexManager controls overall index behavior, and is described in more details in "Managing Indexes" on page 118. When IndexManager is set to autoCommit, it will commit the partially created index, rather than risk running out of resources and failing the index operation.

By default, autoCommit is false. When you send the following message:

```
IndexManager autoCommit: true
```

it configures your IndexManager such that the current transaction is committed during an indexing operation, whenever any of the following occur:

- The current session receives a signal indicating temporary object memory is almost full.
- The percentage of temporary object memory in use reaches the IndexManager's setting for `percentTempObjSpaceCommitThreshold`.

  The default is 60. This threshold can be changed using `IndexManager >> percentTempObjSpaceCommitThreshold:` *anInt*
- The current session receives a signal to FinishTransaction. This occurs when the commit record backlog is larger than STN_SIGNAL_ABORT_CR_BACKLOG, and this session is holding the commit record.
- The number of modified objects in the current transaction reaches the IndexManager's setting for `dirtyObjectCommitThreshold`.

  The default is SmallInteger maximum value, which means this limit is effectively disabled.This limit can be changed using `IndexManager >> dirtyObjectCommitThreshold:` *anInt*

When `autoCommit` is true, a transaction will be started (if necessary) before the indexing operation begins, and the IndexManager will commit at the completion of the indexing operation. Note that this means that, even if you are in manual transaction mode and not in a transaction, index operations will cause changes to be committed to the repository without you explicitly beginning a transaction.

If you want to enable autoCommit only for the current session, not for all index creation, you can use

```
IndexManager sessionAutoCommit: true
```

7.4 Special Kinds of Queries and Indexes

Previously the basic kinds of indexes, on the instance variables of your objects, have been described. The following are some special kinds of indexes, providing some specialized behavior.

Unicode String Indexes and Queries

Equality indexes are inherently ordered. So, while you may search for an object using equality, the ability to find that object using the internal Btree requires that the keys be ordered (collated) in a predicable way with respect to all other keys in that index. The ordering of keys in an equality index is more clear for queries that involve less than and greater than comparisons.

While objects such as numbers and dates have an obvious and fixed ordering, with strings this is more complicated, since different languages may order strings differently. This is described in more detail in Chapter 4.

You may safely create indexes on traditional strings, which have a fixed collation order (you must not change collation using customized Character Data Tables without removing all indexes in the repository).

Unicode strings, which are instance of Unicode7, Unicode16, and Unicode32, allow locale, language, and usage specific collation by relying on instance of IcuCollator. Unicode strings provide a more powerful way to order strings according to the specific requirements for your application. Any ordering that involves a Unicode string needs an IcuCollator, and since the default IcuCollator can change, Unicode strings cannot be used in an ordinary index.

To create an index on a final String element that will permit Unicode strings to be used, you must create a unicode index. A unicode index persists an instance of IcuCollator, which will be used for all comparisons0 within that index. This assures that you can locate elements correctly for a given key, whether it be a unicode string or a traditional string.

NOTE

With Unicode Comparison Mode, these rules change; with this setting, traditional strings behave like Unicode strings in comparison operations. See page 81 for more information.

Creating Unicode Indexes

All unicode indexes require an instance of IcuCollator. An immutable copy of this IcuCollator is persisted as part of the index, and is used for all queries on that index, regardless of the current locale.

If you don’t explicitly specify an instance of IcuCollator, than the current default IcuCollator is used, and will be used for all comparisons on this index.

You cannot change the collator of an index after it has been created.

GsIndexSpec

GsIndexSpec provides special protocol to create unicode indexes. The following methods are available:

```
unicodeIndex:
unicodeIndex:collator:
unicodeIndex:collator:options:
```

If you do not specify a collator, a copy of the current default IcuCollator will be made invariant and persisted with the index. Otherwise, you may specify a collator using standard IcuCollator methods, such as `IcuCollator class >> forLocaleNamed:`. See Chapter 4 for more information on IcuCollator.

UnorderedCollection protocol

You may also create unicode indexes using the traditional UnorderedCollection protocol, by specifying a lastElementClass of any Unicode string class (Unicode7, Unicode 16, or Unicode32).

Since no collator can be specified, the index will be created using the current default IcuCollator.

Due to the way Unicode strings fit into the CharacterCollection hierarchy, the semantics of specifying the lastElementClass are different for CharacterCollection classes, and do not follow the usual hierarchy rules.

- When Unicode7, Unicode 16, or Unicode32 is specified as a lastElementClass, a Unicode index is created, using the current default collator. In addition to allowing instances of Unicode7, Unicode 16, and Unicode32 (regardless of which of these classes is specified), instances of any subclasses of CharacterCollection are allowed.
- When a lastElementClass of String or CharacterCollection is specified, this specifically disallows instances of Unicode7, Unicode16, and Unicode32, although otherwise the hierarchical meaning of the lastElementClass applies.

Example

The following example demonstrates creating a unicode string index, which will collate according to the rules for the German language as used in Germany:

```
GsIndexSpec new
      unicodeIndex: 'each.lastName'
      collator: (IcuCollator forLocaleNamed: 'de_DE');
   createIndexesOn: myEmployees.
```

Queries are created and executed as for equality indexes on ordinary strings. When performing a query, the results are located and ordered according to the collation rules of the IcuCollator that was used to create the index.

For example, since the index was created above with German-language collation, the following query will return results that are correct for German collation:

```
GsQuery
   fromString: '''Weiß''<= each.lastName <= ''Weiz'''
   on: myEmployees
```

Enumerated path terms in indexes and queries

Enumerated path terms allow you query over more than one instance variable value in a single query. This is specified using the vertical bar | in the path term, between the instance variable names.

The instance variables are treated as alternate choices; if any one of the specified instance variables matches the search criteria, the predicate evaluates to true.

For example, you might want to search on both first name and nickname in a single operation. The query might look like this:

```
(GsQuery fromString: 'each.firstName|nickName = ''Freddie'''
   on: MyEmployees) queryResult
```

When this is executed, the results will include all instance that have either the firstName equal to 'Freddie', or the nickName 'Freddie', or both.

In order to optimize this query with an index, you need to create an index on the specific enumeration, e.g. `'each.firstName|nickName'`. An enumerated path term query will not use an index on the individual instance variables that are enumerated.

Restrictions on predicates with enumerated pathTerms

The semantics of enumerated pathTerms do not allow multiple conjoined predicates using the same enumerated pathTerm, since each predicate is evaluated separately. (conjoined predicates are those connected using &).

Collection path Indexes and Queries

Your business objects may themselves contain collections; for example, an employee may contain a collection of children; and you may want to search based on some criteria of the objects in that collection. As long as this collection is itself indexable, indexes and queries can include all elements within these contained collections.

Index paths that include collections, and the queries that use these indexes, are sometimes called Set-valued indexes and queries, although any kind of indexable collection, not just Sets, may be used.

When you wish to specify a path containing an instance of a subclass of UnorderedCollection, the collection is represented by an asterisk *. This syntax may be used to create indexes and perform queries. However, only GsQuery may be used to perform set-valued queries.

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*.

To create the index:

```
GsIndexSpec new
   equalityIndex: 'each.children.*.age'
      lastElementClass: SmallInteger;
   createIndexesOn: myEmployees.
```

Set-valued query results

When you execute a set-valued query, the results you get will follow the particular semantics of Set-valued queries. Since there are potentially multiple "true" query results for a

given element in the base collection, the result of a set-valued query such as this can be larger than the original collection.

For example, consider the following query, using the index created above:

```
(GsQuery fromString: 'each.children.*.age <= 18'
   on: myEmployees) queryResults
```

In this example, if the root collection myEmployees is a Bag or IdentityBag (rather than a Set or IdentitySet), and an employee has two children that are under 18, then that employee will appear in the results (a Bag or IdentityBag) twice. Employees with three minor children appear in the results three times, and so on. The resulting collection may be several times as large as the original collection, depending on the details of the query and data.

If the root collection myEmployees is a Set, which does not allow multiple instances of the same object, this potential source of confusion does not occur.

Restrictions on predicates in set-valued queries

The semantics of set-valued indexes do not allow multiple conjoined predicates that use the same set-valued pathTerm, since each predicate is evaluated separately. (conjoined predicates are those connected using &).

In general, it is recommended to avoid using multiple- set-valued predicate queries, although some multiple-predicate set-valued queries can be optimized, or avoid the problem cases, and are safe and therefor allowed.

Redefined Comparison Messages

When indexed queries are executed for instances of basic classes or subclasses of basic classes (see "Basic Classes optimized for indexes" on page 98), the comparison operators are not performed as message sends, and you cannot change the operation of a query 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 or subclassed the class Time, redefining < to count backwards from the end of the century, GemStone Smalltalk would ignore that redefinition when < appeared next to an instance of Time inside a selection block. GemStone Smalltalk would simply apply an operator that behaved like Time's standard definition of <.

For subclasses that you have created that are not subclasses of basic classes, 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.

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` so that dictionaries will behave in a consistent and logical manner. This rule is not limited to indexes, but applies to any object that will be stored in any dictionary.

WARNING: Modifying an existing object in such a way that its hash value changes will corrupt hashed collections containing that object. Use care if you need to redefine the hash method; do not refer to instance variables that are likely to change.

7.5 Managing Indexes

You may need to find out about all the indexes in your system, and to remove selected indexes or clean up indexes that were not successfully created. This functionality is provided by the class IndexManager.

IndexManager has a single instance which provides much of the functionality, accessible via:

```
IndexManager current
```

This instance is lazy initialized, and stored in the IndexManager class instance variable after it is created. Any configuration you do on IndexManager current, therefore, will be used by all affected operations, if you commit after making the change.

Indexes on temporary collections

You may create indexes on temporary collections containing temporary and persistent objects. However, on abort, any indexes on temporary collections are removed.

Inquiring About Indexes

For a full description of the indexes on a particular collection, send `indexSpec` to the collection. This produces a string containing the GsIndexSpec code that would recreate the same indexes, and provide useful documentation on those indexes.

For example,

```
myEmployees indexSpec
%
GsIndexSpec new
   equalityIndex: 'each.age'
      lastElementClass: SmallInteger;
   equalityIndex: 'each.address.state'
      lastElementClass: String;
      options: GsIndexOptions reducedConflict;
   identityIndex: 'each.userId';
   yourself.
```

You can also send messages to the collection that will return quick information on indexed paths.

- `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.

  For example, the following expression returns the paths into myEmployees that bear equality indexes:

  ```
  myEmployees equalityIndexedPaths
  %
  anArray( 'age', 'address.state')
  ```

- `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 specified when creating an index using the keyword `withLastElementClass:`.

The following IndexManager messages allow you to inquire about all indexes in the repository.

- `getAllNSCRoots`

  Returns a collection of all UnorderedCollections in the repository that have indexes.

- `usageReport`

  Returns a report on all indexes on all UnorderedCollections in the repository.

Removing Indexes

There are a number of ways to remove indexes, using GsIndexSpec, IndexManager, and UnorderedCollection protocol.

Since indexing internal structures create references to the indexed collection and to objects in the collection, before dereferencing a collection, you should be sure to remove all indexes on the collection. This allows the collection to be garbage collected.

To remove indexes based on a GsIndexSpec

As you can create indexes based on an instance of GsIndexSpec, you can also use that specification to remove these indexes.

`GsIndexSpec >> removeIndexesFrom:` *aCollection*
Removes the indexes described by the receiver from the collection indicated by *aCollection*. If any specified indexes do not exist, they are not removed and no error is returned.

This is most useful in combination with the method that creates the spec from the existing collection. For example:

```
(MyEmploees indexSpec)
   removeIndexesFrom: MyEmployees.
```

To remove a single index, you may edit the specification code printed by `indexSpec`, or create a simple GsIndexSpec with information to remove a single index:

```
(GsIndexSpec new
   equalityIndex: 'each.age' lastElementClass: Object)
      removeIndexesFrom: MyEmployees.
```

To remove indexes using IndexManager

IndexManager, which provides a system-wide view of all the indexes in the repository, provides a number of methods to remove indexes both individually, by collection, and globally.

`IndexManager >> removeEqualityIndexFor:` *aCollection* `on:` *aPathString*
Removes an equality index from the collection *aCollection* with the indexed path described by *aPathString*. If the path specified does not exist, this method returns an error. Implicit indexes are not removed.

`IndexManager >> removeIdentityIndexFor:` *aCollection* `on:` *aPathString*
Removes the identity index from the collection *aCollection* with the indexed path described by *aPathString*. If the path specified does not exist, this method returns an error. Implicit indexes are not removed.

`IndexManager >> removeAllIndexesOn:` *aCollection*
Removes all explicitly created indexes from the collection *aCollection*. Implicit indexes that were created by these elements participating in other indexed collections are not removed.

`IndexManager >> removeAllIndexes`
Removes all indexes on all UnorderedCollections, including all implicit and partial indexes.

`IndexManager >> removeAllTracking`
Removes all indexes on all UnorderedCollections, and all object tracking. While this is the fastest way and most complete way to remove indexing infrastructure, if you are using modification tracking for any other purpose, that tracking will be removed as well.

To remove indexes using UnorderedCollection protocol

You may also send methods to the indexed collection directly to remove one or all indexes.

`UnorderedCollection >> removeEqualityIndexOn:` *aPathString*
Removes an equality index from the path indicated by *aPathString*. If the path specified does not exist, this method returns an error. Implicit indexes are not removed.

`UnorderedCollection >> removeIdentityIndexOn:` *aPathString*
Removes the identity index on the specified path. If the path specified does not exist, this method returns an error. Implicit indexes are not removed.

`UnorderedCollection >> removeAllIndexes`
Removes all explicitly created indexes from the receiver. Implicit indexes that were created by these elements participating in other indexed collections are not removed.

Rebuilding Indexes

When objects that participate in an index are modified, the related indexing infrastructure must be updated. This causes some overhead. If you are performing an operation that will modify a large number of objects that participate in multiple indexes, such as a large migration, it may be more efficient to remove some or all of the indexes on the collection before performing the migrate, and rebuild those indexes after the migration is complete.

It is also sometimes required to remove and rebuild indexes as part of a GemStone upgrade; certain changes in GemStone kernel classes require you to either rebuild specific kinds of, or all, indexes. Any requirement to do this will be included in upgrade instructions in the *Installation Guide* for the version of GemStone to which you are upgrading.

To remove and rebuild indexes, you can extract and save the GsIndexSpec, and reuse that after the operation is complete.

For example:

```
| mySpec |
mySpec := myCollection indexSpec.
mySpec removeAllIndexesFrom: myCollection.
<perform migration or other operation>
mySpec createIndexesOn:myCollection
```

Using `IndexManager >> getAllNSCRoots`, you may extend this example to retrieve the GsIndexSpec for each collection in the repository, which will allow you to remove and rebuild the indexes.

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.

Formulating queries and performance

The most efficient queries are the ones in which the first predicate will return the smallest result set. This is sometimes easy for a human to determine, but the query cannot predict this without actually running the query. Queries should be manually reviewed for these kinds of domain-specific optimizations.

For example, you might want to query for current orders for a particular customer.

```
(each.status = #current) & (each.customer.name = 'Smith')
```

If your application is likely to have only a few current orders, then this is more efficient. However, if you are likely to have many current orders, but only a few customers named Smith, it would be more efficient for you to write the formula in reverse order.

This assumes that both predicates have an associated index. The optimization step will reorders predicates so the indexed predicates will be evaluated before any non-indexed predicates. See "Query Formula Optimizations" on page 126 for the automatic optimizations that are done.

Indexing Errors

To ensure that indexing structures are consistent, some kinds of errors that may occur during index creation will disable commits. Before creating an index, it is advisable to commit any work in progress, to avoid losing any work if an indexing error does occur.

For example, if you create an index on a collection and one or more of the objects that participate in the index do not implement the instance variable on the path, it will raise an error (unless using optionalPathTerms, as described starting on page 110).

If an error occurs partly through index creation, and the autoCommit status (see page 113) means that some portion of the index creation was committed, a collection may have unusable partial indexes. These indexes must be manually removed.

The following IndexManager instance methods allow you to remove incomplete indexes, while not affecting any complete, usable indexes:

`IndexManager current removeAllIncompleteIndexes`
Removes all incomplete indexes on all UnorderedCollections.

`IndexManager current removeAllIncompleteIndexesOn:` *anNSC*
Removes all incomplete indexes on the specified UnorderedCollection.

If you modify objects that participate in an index, try to commit your transaction, and your commit operation fails, query results can become inconsistent. If this occurs, abort the transaction and try again.

Auditing Indexes

Indexes should be audited regularly, as part of your regular application maintenance, to ensure there are no problems.

You can audit the internal indexing structures for a particular collection by executing:

aCollection `auditIndexes`

This audits all the indexes, explicit and implicit, on the given collection. If indexes are correct, this method returns 'Indexes are OK' or 'Indexes are OK and the receiver participates in one or more indexes.'. If there are no indexes on the collection, a message such as 'No indexes are present.' is returned.

In the case of failure, a list of specific problems is returned.

You can audit all indexes in the entirely repository at once using:

```
IndexManager current nscsWithBadIndexes
```

which will return an IdentitySet containing all collections that fail auditIndexes. Depending on the number of indexed collections in your system, this may take a considerable time to run.

In the rare case of a problem reported, the usual way to resolve the problem is to remove and rebuild the affected indexes. In some cases, removing all indexes on the collection may succeed even if the internal problems prevent a single index being removed. If removing indexes is impractical, contact Gemstone Technical Support for further assistance.

7.6 Query Formulas and Optimization

When you define a query, you may be able to most easily write this as a statement of business logic. For more complicated queries, this may not be in its most efficient form.

Automatic query optimization performs some optimizations that can change a query into the logically equivalent form that is more efficient for GemStone to execute.

You can also perform these optimizations manually, by writing your query in a the most efficient form, rather than in the human/business logic form.

Query Formulas

An instance of GsQuery is created on a string, and is internally represented as an instance of GsQueryFormula. The formula provided by the string may not be in its most efficient form for execution. While you can hand-optimize the formula when you create the string, it may be desirable to write the query so it makes sense from a business logic point of view, and is more human-readable.

While a query formula such as

```
each.numberOfChildren < 3
```

does not change from optimization, a more complicated query such as this:

```
((1 <= each.numberOfChildren) not &
   (each.numberOfChildren <= 3)) not
```

benefits from optimization; the optimized version of this is

```
each.numberOfChildren < 1
```

When a query is being executed or displayed, by default it is auto-optimized. It uses the current formula to create an optimized version according to the optimization rules, and executes or displays the optimized formula. The optimized version is not saved.

You can access the current formula using the `formula` message. By accessing the instance of GsQueryFormula directly, you bypass the auto-optimization.

You can update the query's formula by sending the `optimize` message to the query. This saves the new, optimized formula in place of the current formula in the query.

Queries also retain the original formula with which they are created, so you can still view the human-readable form. These are accessible via the `originalFormula` message.

For example, if you create a query based on the above formula:

```
query := GsQuery fromString: '((1 <= each.numberOfChildren) not
   & (each.numberOfChildren <= 3)) not'.
```

You can view the formulas, before and after optimization:

```
query printString
each.numberOfChildren < 1

query formula printString
((1 <= each.numberOfChildren) not & (each.numberOfChildren <=
3)) not

query optimize.
query formula printString
each.numberOfChildren >= 1

query originalFormula printString
((1 <= each.numberOfChildren) not & (each.numberOfChildren <=
3)) not
```

Invariance and Formula reuse

Instances of GsQuery are not invariant. However, instances of GsQueryFormula and its subclasses are invariant when created using the public API. This allows formulas to be safely persisted and shared, since side effects of message sends will not change the semantics of the query.

When a GsQuery's formula changes, such as when variables are bound, or when `optimize` is sent, a new formula instance replaces the previous one. The particular formula in a GsQuery, therefore, will depend on the stage at which the formula is accessed.

For example, to save and reuse a formula from a GsQuery:

```
aQuery := GsQuery fromString: 'each.age <= min'.
UserGlobals at: #savedFomula put: aQuery formula.
```

The formula can be later reused:

```
(GsQuery fromFormula: savedFomula on: aCollection)
   bind: 'min' to: 18;
   queryResults.
```

To make a GsQuery invariant so it can safely be reused, send `immediateInvariant`. The query can later be copied, to bind, optimize and execute. In this example, note that the query is saved with the reference to the collection:

```
query := GsQuery fromString: 'each.age <= min' on: aCollection.
UserGlobals at: #savedQuery put: query immediateInvariant.
```

In this case, the reused query does not need to specify the collection when reused:

```
savedQuery copy
   bind: 'min' to: 18;
   queryResults.
```

Disabling auto-optimize

Queries, by default, are optimized before execution. Each query has an associated instance of GsQueryOptions. This controls optimization and other query features. In addition to the various specific optimizations performed, GsQueryOptions controls if automatic query optimization is done. The default is to do auto-optimization.

Queries are created with a default GsQueryOptions, or the options can be set on creation using the query creation methods with the options: keyword.

To disable auto-optimization, you can create a query that specifies an instance of GsQueryOptions that has autoOptimize removed. For example:

```
query := GsQuery
   fromString: '((1 <= each.numberOfChildren) not &
         (each.numberOfChildren <= 3)) not'
   on: myEmployees
   options: (GsQueryOptions default - GsQueryOptions
         autoOptimize).
```

Query Formula Optimizations

The following are the specific optimizations that are performed when a query is optimized.

Remove "not" using boolean logic

Clauses that include a `not` are transformed using De Morgan's Laws into the logical equivalent form without the not. For example:

```
(each.firstName = 'Dale') not
```

becomes:

```
(each.firstName ~= 'Dale')
```

Convert predicates with equal operands into boolean constants

Predicates with common operands are transformed to the equivalent constant predicate. For example, either of the following becomes `true`:

```
(each.firstName = each.firstName)
(4 = 4)
```

Convert constant-path reversed to path-constant

Constant-path predicates are replaced by equivalent path-constant predicates. For example:

```
(19 > each.age)
```

becomes:

```
(each.age < 19)
```

Eliminate redundant predicates

Predicates that fall within range of other predicates are removed. For example:

```
(each.age < 19) & (each.age < 4)
```

becomes:

```
(each.age < 4)
```

This optimization requires that the variables in the query be bound to values.

Combine path-constants into range predicate

Two path-constant predicates on the same path will be converted into a single range predicate, if the predicates represent a range. For example:

```
(each.age > 4) & (each.age < 19)
```

becomes:

```
(4 < each.age < 19)
```

This optimization requires that the variables in the query are bound to values.

Combine path-constants to enumerated predicate

If an index exists that has an enumerated path term, and there are path-constant predicates using that enumerated path term, the path-constant predicates can be combined into a single enumerated predicate. For example,

```
(each.firstName = 'Martin') | (each.lastName = 'Martin')
```

becomes:

```
(each.firstName|lastName = 'Martin')
```

This optimization requires that the collection and any variables be bound to the query.

Simplify (true) and (false) predicates

Other optimizations may result in predicates that are unary constants `true` or `false`. These are removed, or the entire expression is simplified, depending on the logic.

`(true) & <other predicates>` becomes: `<other predicates>`

`(true) | <other predicates>` becomes: `(true)`

`(false) & <other predicates>` becomes: `(false)`

`(false) | <other predicates>` becomes: `<other predicates>`

Reorder predicates

The predicates are reordered as follows, from left to right.

1. predicates involving indexed paths.
2. predicates with identity comparisons on paths without indexes.
3. predicates with equality comparisons on paths without indexes.

Chapter

8 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.

How GemStone Detects and Manages Conflict
describes how commit conflicts are detected and reported and how to handle and avoid conflicts.

Controlling Concurrent Access with Locks
discusses the kinds 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.

8.1 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.

Views and Transactions

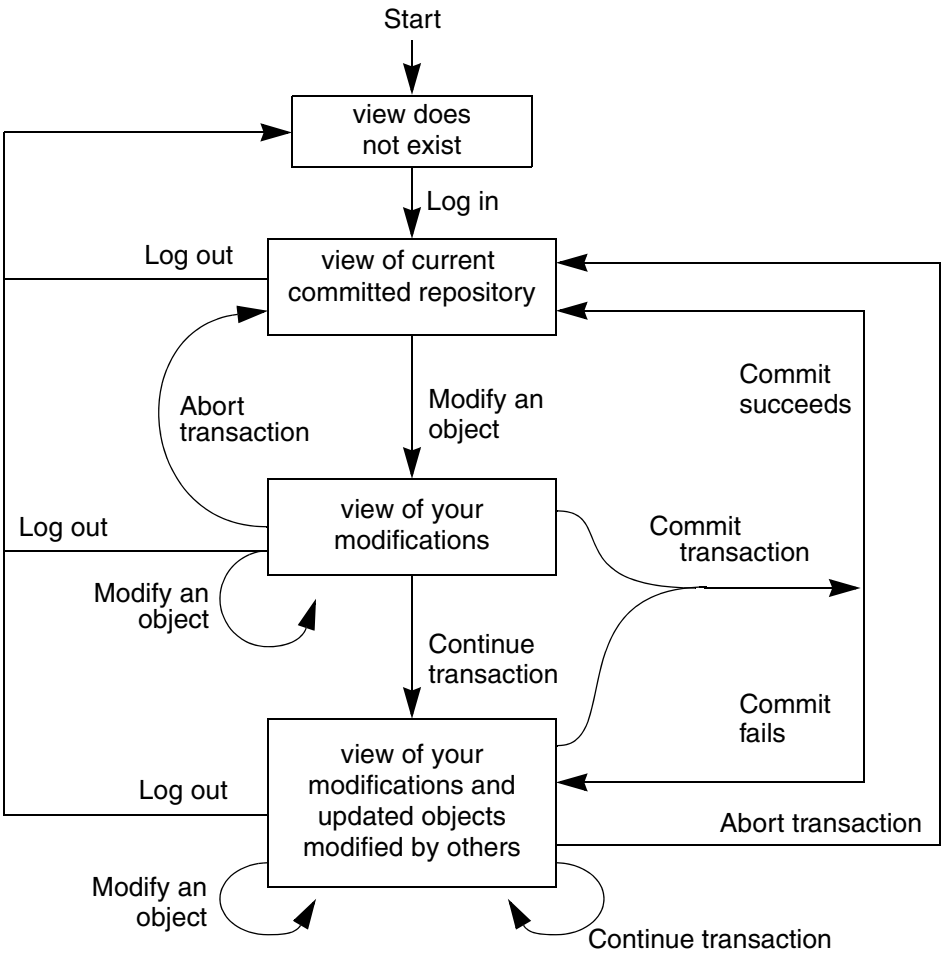
As shown in Figure 8.1, every user session maintains its own consistent view of the repository state. Objects that the repository contained at the beginning of your session are preserved in your view, even if you are not using them—and even if other users' actions have rendered them obsolete. The storage that those objects 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 and the commit frequency), this burden can be trivial or significant.

When you log in to GemStone, you get a view of repository state. After login, you may start a transaction automatically or manually, or remain outside of transaction. The repository view you get on login is updated when you begin a transaction or abort. When you commit a transaction, your changes are merged with other changes to the shared data in the repository, and your view is updated. When you obtain a new view of the repository, by commit, abort, or continuing, any new or modified objects that have been committed by other users become visible to you.

The transaction mode controls if a transaction is automatically began, or if you must manually begin a transaction. For details, see "Committing Transactions" on page 135.)

Figure 8.1 View States



Transaction State and Transaction Modes

A GemStone session is always either in a transaction or not in a transaction. When in transaction, changes can be committed to the repository. When not in transaction, you can make changes in your view but these changes cannot be committed.

A session that is in transaction may be in one of a number of transaction levels, depending on if nested transactions are involved.

When not in transaction, the session may merely be not in transaction, or it may be in the specialized transactionless mode. In transactionless mode, the session is not in transaction, but its view may be updated automatically at any time. Transactionless mode is primarily for idle sessions that do not need a reliable view of repository data; the topics that this chapter discusses for the most part do not apply to transactionless mode sessions.

The transaction modes provide different behavior with respect to starting new transactions. When in automatic transaction mode, the session is always in transaction. When in manual transaction mode, you may be in transaction or not in transaction, depending on specific messages your session sends.

The following are the GemStone transaction 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. Use caution with this mode in busy production systems, since your session will not receive the signals that your view is causing a strain on system resources.

This is the default transaction mode on login.

To change to transactionless transaction mode, send the message:

```
System transactionMode: #autoBegin
```

This aborts the current transaction and starts a new transaction.

Manual transaction mode

In this mode, you can be logged in and outside of a transaction. You explicitly control whether your session starts a transaction, makes changes, and commits. Although a transaction is started for you when you log in, you can set the transaction mode to manual, which aborts the current transaction and leaves you outside a transaction. You can subsequently 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 manual transaction mode, you can view the repository, browse objects, and make computations based upon object values. You cannot, however, make your changes permanent, nor can you add any new objects you may have created while outside a transaction. You can start a transaction at any time during a session; you can carry temporary results that you may have computed while outside a transaction into your new transaction, where they can be committed, subject to the usual constraints of conflict-checking.

To change to manual transaction mode, send the message:

```
System transactionMode: #manualBegin
```

This aborts the current transaction and leaves the session not in transaction.

To begin a transaction, execute

```
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.

Transactionless mode

In transactionless mode, you remain outside a transaction. This mode is intended primarily for idle sessions. If all you need to do is browse objects in the repository, transactionless mode can be a more efficient use of system resources. However, you are at risk of obtaining inconsistent views.

To change to transactionless transaction mode, send the message:

```
System transactionMode: #transactionless
```

Determining transaction mode and transaction state

To determine the transaction mode you are in, send the message:

```
System transactionMode
```

To determine the transaction level you are at, send the message:

```
System transactionLevel
```

To determine if you are in transaction, send the message

```
System inTransaction
```

A transaction level of 1 or more means your session is in transaction, with values greater than 1 indicating the number of levels of transaction. A transaction level of 0 is not in transaction, while -1 indicates transactionless.

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.

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 security policy 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 same operation that anEmployee is written:

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

Some less common operations cause objects to be read or written. For example, assigning an object to a new object security policy, using the message `assignToObjectSecurityPolicy:`, writes the object and reads both the old and the new GsObjectSecurityPolicy. 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.

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.

Nested In-memory Transactions

Within a transaction, GemStone allows you to group units of work into logical transactions, which can be committed or aborted within the given session. These logical transactions can be nested with up to 16 levels of nesting (including the outer level actual transaction). When the full set of changes are ready to be committed, committing the outer transaction will make the changes persistent and detect any conflicts.

While the same protocol is used to commit the actual (outer) transaction and the nested transactions, the semantics are different. A commit of a nested transaction does not detect conflicts with changes by other users, does not update current session state, and does not make the changes persistent if the session exits unexpectedly or recoverable on system shutdown. Abort of a nested transaction returns the session to the state it was in at the beginning of the nested transaction, without updating the session's view with any changes by other users.

When transactions are discussed, unless specified otherwise, it only refers to an outer level actual transaction, not to a nested transaction.

To begin a nested transaction, use

```
System beginNestedTransaction
```

You should be already in transaction when executing this method.

Executing `commit`, `commitTransaction`, `abort`, or `abortTransaction` when in a nested transaction preserve or discard in-memory changes and return to the parent level of transaction. The same protocol is used at the outer level, actual transaction to perform the commit or abort.

`continueTransaction` cannot be used when in a nested transaction.

You can commit or abort all levels of nested transactions at once, including performing the outer level actual commit or abort, using the messages:

```
System commitAll
System abortAll
```

8.2 How GemStone Detects and Manages 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 also in another session's dependency list—a *write-dependency conflict*. An object belongs to a session's *dependency list* if the session has added, removed, or changed a dependency (index) for that object. For details about how GemStone creates and manages indexes on collections, see Chapter 7, Indexes and Querying.

If a write-write or write-dependency conflict is detected, then your transaction cannot commit. 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.

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 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 *reduced-conflict (RC) 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 include: RcCounter, RcIdentityBag, RcQueue, and RcKeyValueDictionary. See "Classes That Reduce the Chance of Conflict" on page 150.

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 performs these actions:

1. Checks whether other concurrent sessions have committed transactions that modify an object that you modified during your transaction.
2. Checks to see whether other concurrent sessions have committed transactions of their own, modifying an object that you have read during your transaction, while at the same time you have modified an object that another session has read.
3. Checks to see whether other concurrent sessions have added, removed, or changed indexes on an object that you have modified during your transaction.
4. 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 messages `commit` or `commitTransaction` commit the current transaction:

Example 8.1

```
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. The message `commit` performs the same commit, but returns true if GemStone commits your transaction and signals an error if it fails to commit.

To find why your transaction failed to commit, you can send the message:

```
System transactionConflicts
```

This method returns a symbol dictionary that contains an Association whose key is #commitResult and whose value is one of the following symbols:

#readOnly
#success
#rcFailure
#dependencyFailure
#failure
#retryFailure
#commitDisallowed
#retryLimitExceeded

The remaining Associations in the dictionary are used to report the conflicts found. Each Association's key indicates the kind of conflict detected; its associated value is an Array of OOPs for the objects that are conflicting.

Table 1 lists the possible keys for the conflict.

Table 1 Transaction Conflict Keys

Key	Meaning
#'Read-Write'	StrongReadSet and WriteSetUnion conflict. Used by GemStone indexing mechanism.
#'Write-Write'	WriteSet and WriteSetUnion conflict.
#'Write-Dependency'	WriteSet and DependencyChangeSetUnion conflict.
#'Write-WriteLock'	WriteSet and WriteLockSet conflict.
#'Rc-Write-Write'	Logical Write-Write conflict on an instance of a reduced conflict class.

If there are no conflicts for the transaction, the returned symbol dictionary has no additional Associations.

Conflict sets are cleared at the beginning of a commit or abort and thus can be examined until the next commit, continue, or abort.

NOTE

To avoid making conflict sets persistent, be sure to disconnect them before committing.

To determine whether the current transaction has write-write conflicts, you can send the following message before attempting to commit the transaction:

```
System currentTransactionHasWWConflicts
```

Similarly, to determine whether the current transaction has write-dependency conflicts, you can send this message:

```
System currentTransactionHasWDConflicts
```

If the above message returns true, you can send the appropriate message to obtain a list of write-write (or write-dependency) conflicts in the current transaction:

`System currentTransactionWWConflicts` (write-write)

or:

`System currentTransactionWDConflicts` (write-dependency)

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 (or involved in indexing operations) 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.

Indexes and Concurrency Control

It is also 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. It is true even if the collection itself is an Rc collection and does not encounter transaction conflicts.

To check this possibility, examine the dictionary returned by evaluating `System transactionConflicts` (page 136). If that dictionary includes any Associations whose key is #'Write-Dependency', 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.

If you encounter conflicts in the internal indexing structures, you can create a reduced-conflict index. See "Reduced-Conflict Indexes" on page 110.

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 messages `abort` or `abortTransaction` discard the modified objects in your view. If you are in automatic transaction mode, these messages also begin a new transaction.

Example 8.1

```
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 within your own object space 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—because the transaction was aborted, your changes did not affect objects in the repository. The new view, however, does include changes committed by other users since your last transaction started. Objects that you have created in the GemBuilder for Smalltalk object space, outside the repository, 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 if the repository has changed.

Unlike `commitTransaction` and `abortTransaction`, `continueTransaction` does not end your transaction. It has no effect on object locks, and it does not discard any changes you have made or commit any changes. Objects that 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.

Note that if you were unable to commit your transaction due to conflicts, you cannot use `continueTransaction` until you abort the transaction.

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 before it attempts to reclaim obsolete objects in your view. While you are in a transaction, your session will not be signalled to abort, nor is it subject to losing it's view of the repository or being terminated

when it does not respond to a signal to abort. However, a session in transaction may cause your repository to grow until it runs out of disk space.

When you are outside of a transaction, GemStone warns you when your view is outdated, and keeping it available for you is imposing a burden on the system, by sending your session the `TransactionBacklog` notification. 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. 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 sends your session the exception `RepositoryViewLost` and then either terminates the Gem or forces an abort, depending on the value of the related configuration parameter STN_GEM_LOSTOT_TIMEOUT. (These parameters are described in Appendix A of the *System Administration Guide*.) Forcing an abort recomputes your view of the repository; copies of objects that your application had been holding may no longer be valid.

Work that 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 guarantees that the application values are valid.

Your GemStone session controls whether it is signalled to abort by receiving the TransactionBacklog notification when it is out of transaction. To enable receiving it, send the message:

```
System enableSignaledAbortError
```

To disable receiving it, send the message:

```
System disableSignaledAbortError
```

To determine whether receiving this notification is currently enabled or disabled, send the message:

```
System signaledAbortErrorStatus
```

This method returns true if the notification is enabled, and false if it is disabled. By default, GemStone sessions disable receiving this notification. The GemBuilder interfaces may change this default. If you wish to be notified, then you must explicitly enable the signaled abort error, and re-enable it after each time the signal is received.

Being Signaled to continueTransaction

As described earlier, when you are in a transaction, GemStone does not signal the session to abort, nor are you subject to losing your view of the repository. This entails a risk that your repository may grow until it runs out of disk space.

To avoid this problem, you can enable your GemStone session to receive the `TransactionBacklog` notification when you are in transaction. This prompts your session that it is now holding the oldest view of the repository, and potentially causing your repository to grow. When your session receives this signal, it may execute a `continueTransaction`, or abort or commit its changes.

Your GemStone session controls whether it receives the `TransactionBacklog` notification when in transaction. To enable receiving it, send the message:

```
System enableSignaledFinishTransactionError
```

To disable receiving it, send the message:

```
System disableSignaledFinishTransactionError
```

To determine whether receiving this error message is currently enabled or disabled, send the message:

```
System signaledFinishTransactionErrorStatus
```

This method returns true if the notification is enabled, and false if it is disabled. By default, GemStone sessions disable receiving this notification. If you wish to be notified, then you must explicitly enable it after each time the signal is received.

Handlers for abort or continueTransaction notifications

Not only do you need to enable the receipt of the notification to abort or continueTransaction, you must also set up a signal handler to take the appropriate action. Sending `enableSignaledAbortError` and `enableSignaledFinishTransactionError` control whether you receive the TransactionBacklog notification when you are not in transaction or when you are in transaction, respectively. The handler for the TransactionBacklog notification needs to take both possible situations into account.

8.3 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, regardless of your transaction mode or whether you are in a transaction. When you are in 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 request a lock when outside a transaction.

Lock Types

GemStone provides two kinds of locks you may use on any objects: *read* and *write*. A session may hold only one kind of lock on an object at a time. GemStone also provides another type of lock, *applicationWriteLock*, which is limited to a single unique lock object; it behaves similarly but is used to provide a mutex. While these behave similarly to read and write locks, they are used differently and are discussed separately.

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 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. Because locking incurs a cost at commit time, you should keep the aggregate number of locked objects as small as possible.

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 either a read or write 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.

Acquiring Locks

The kernel class System is the receiver of all lock requests. The following statements request one lock of each kind:

Example 8.2

```
System readLock: SharedDictionary.
System writeLock: myEmployees.
```

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.)

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 require write authorization.
- The object is an instance of SmallInteger, Boolean, Character, SmallDouble, or nil. Trying to lock these special objects is meaningless.
- The object is already locked in an incompatible way by another session (remember, only read locks can be shared).

Variants of the `readLock:` and `writeLock:` messages allow you to lock collections of objects en masse. For details, see "Locking Collections of Objects Efficiently" on page 144.

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 this:

```
System readLock: anObject
   ifDenied: [block1]
   ifChanged: [block2].
```

A lock denial causes GemStone to execute the block argument to `ifDenied:`. The method in Example 8.3 uses this technique to request a lock repeatedly until the lock becomes available.

Example 8.3

```
testObject := Object new.
%
Object subclass: #Dummy
```

```
   instVarNames: #()
   classVars: #()
   classInstVars: #()
   poolDictionaries: #()
   inDictionary: UserGlobals
   options: #()
%
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: testObject
%
method: Dummy
getReadLockOn: anObject
System readLock: anObject
       ifDenied: [self getReadLockOn: anObject]
       ifChanged: [System abortTransaction]
%

Dummy new getReadLockOn: testObject
%
```

Dead Locks

You may never succeed in acquiring 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 8.4

```
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. See Example 8.5.

Example 8.5

```
| 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 messages to request locks on an entire collection 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 to use the collection-locking methods than to lock the objects individually.

The following statements request locks on each of the elements of two different collections:

Example 8.6

```
UserGlobals at: #myArray put: Array new;
   at: #myBag put: IdentityBag new.
```

```
System readLockAll: myArray.
System writeLockAll: myBag.
```

The messages in Example 8.6 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 GemStone 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 array contains all elements for which dirty locks were granted.
- The third array is empty, and is there for compatibility with previous versions of GemStone.

You can then take appropriate actions within the block. See Example 8.7.

Example 8.7

```
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 read lock to a write lock. For example, you might initially intend to read an object, only to discover later that you must also write the object.

However, if you have a read lock on an object, you cannot successfully write that object. If you attempt to do so, your attempt to commit that transaction will fail.

GemStone currently provides no built-in support for upgrading locks. However, to ensure your ability to commit, you can remove the read lock you currently hold on an 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 discussed in the section “Indexes and Concurrency Control” on page 137. 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 the following messages for removing locks:

`System removeLock:` *anObject*
: Removes any lock you might hold on a single object. 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.

`System removeLockAll:` *aCollection*
: Removes any locks you might hold on the elements of a collection.

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. Class System provides two mechanisms for doing so.

`System commitTransaction; removeLocksForSession`
: Attempts to commit the present transaction and removes all locks it holds, even if the commit does not succeed.

`System commitAndReleaseLocks`
: Attempts to commit your transaction and release all the locks you hold in a single operation. 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 objects whose locks you wish to release as soon as you commit your current transaction. The other set contains 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 element is always empty)

The following code uses this information to remove all write locks held by the current session:

```
System removeLockAll: (System sessionLocks at: 2)
```

Another useful 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 that 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 Example 8.8 uses `lockOwners:` to build an array of the userIDs of all users whose sessions hold locks on a particular object.

Example 8.8

```
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 *anObject* 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 *anObject* 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 image.

Application Write Locks

Unlike read and write locks, application write locks can only be placed on a single object per lock queue (there are two lock queues available). The object can be any persistent object; the first time an application lock write is invoked on a lock queue, the object that is locked is registered for that lock queue, and all subsequent uses of that lock queue can only lock this particular object until the next Stone restart.

This allows it to be used as a mutex, or simplifies serializing modifications to a single critical object, such as a collection.

The other difference in locking behavior is that invoking the method to place an application write lock does not return until the lock is acquired, or the lock wait times out.

The timeout is controlled by the configuration parameter STN_OBJ_LOCK_TIMEOUT. This frees you from having to repeatedly request a lock if it is not immediately available.

To set an application write lock on an object, send the message:

```
System waitForApplicationWriteLock: lockObject queue: lockIdx
autoRelease: aBoolean
```

lockId must be 1 or 2, depending on which lock queue is being used.

If *aBoolean* is true, the lock is released automatically on commit or abort, otherwise you must manually remove the lock when you are done.

This method returns an integer code, one of the following:

1 - lock granted
2071 - undefined lock (lockIdx out of range or lockObject is special object)
2074 - dirty; the lock object written by other session since start of this transaction
2418 - lock not granted, deadlock
2419 - lock not granted, wait for lock timed out

8.4 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
- 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, 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 significantly reduces write-write conflicts on their instances.

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 8.9

```
!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 of these conditions exists:

- 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.
- 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.

Indexing an instance of RcIdentityBag does diminish somewhat its "reduced-conflict" nature, because of the possibility of a conflict on the underlying indexing structure. (For a more complete explanation of this possibility, see "Indexes and Concurrency Control" on page 137.) You can reduce the risk further by using reduced conflict equality indexes; see

"Creating Indexes" on page 107. 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 of these conditions exists:

- 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 way 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.

To remove obsolete entries belonging to all inactive sessions, the producer can send the message `cleanupQueue`.

You may also experience commit conflicts when additional users begin to add or remove objects from the RcQueue, since the internal structure of the RcQueue itself is not reduced-conflict. If you know in advance how many users will be adding or removing

from the RcQueue, you should specify the RcQueue size on creation using the `new:` method.

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 of these conditions exists:

- 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.
- Any number of users perform any combination of these operations.

Chapter

9 Object Security and Authorization

This chapter explains how to set up object security policies to restrict read and write access to application objects. It covers:

How GemStone Security Works
describes the Gemstone object security model.

Assigning Objects to Security Policies
summarizes the messages for reporting your current security policy, changing your current policy, and assigning a policy to simple and complex objects.

An Application Example and A Development Example
provides examples for defining and implementing object security for your projects.

Privileged Protocol for Class GsObjectSecurityPolicy
defines the system privileges for creating or changing security policy authorization.

9.1 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: GemBuilder for Smalltalk, GemBuilder for Java, Topaz, 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 (depending on the login authentication used) your 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, or objects, that the user has access to—UserGlobals, Globals, and Published) for resolving symbols when compiling. Chapter 3, "Resolving Names and Sharing Objects," discusses these topics.
- The groups to which you belong and any special system privileges you may have.
- A default GsObjectSecurityPolicy to assign your session at login, or nil.

See the *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, GsObjectSecurityPolicy, and Repository classes, and to modify code. 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, GsObjectSecurityPolicy and Repository Classes are described in the image, and their use is discussed in the *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. Objects can also be left unsecured, so any user can read or modify them. Not restricting access will improve performance for sites with fewer security requirements.

The GemStone class GsObjectSecurityPolicy facilitates this security.

GsObjectSecurityPolicy

Each object's header includes a 16-bit unsigned security policy Id that specifies the GsObjectSecurityPolicy to which the object has been assigned. In previous releases, object security policies were known as Segments. In GemStone/S 64 Bit 3.0, Segment has been renamed to GsObjectSecurityPolicy, to more clearly represents its function. All references to Segment in previous releases now pertain to GsObjectSecurityPolicy.

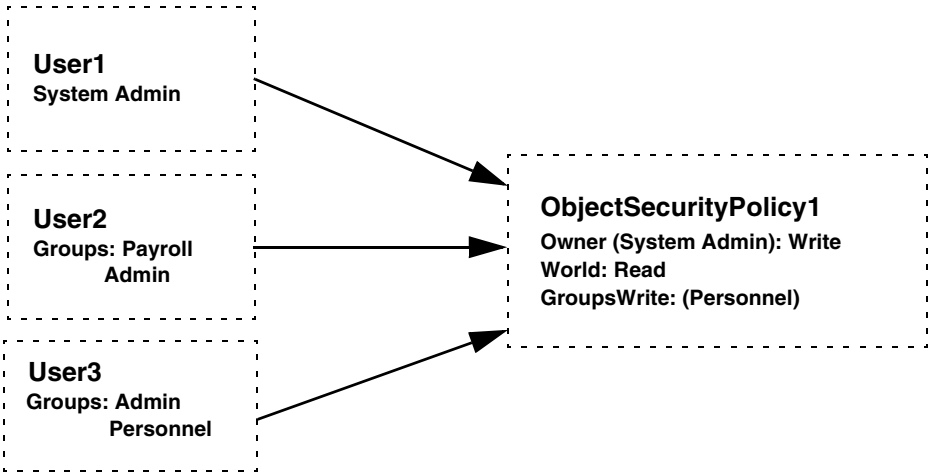
All objects assigned to an security policy have exactly the same protection. That is, if you can read or write one object assigned to a certain policy, you can read or write them all. Each policy is owned by a single user, and all objects assigned to the same security policy have the same owner. Groups of users can have read, write, or no access to an security policy. Likewise, any authorized GemStone user can have read, write, or no access to a policy.

An object may also have no security policy, in which case its security policy Id is zero. This means that there are no restrictions on access to this object; any logged-in user can read and write this object.

Whenever an application tries to access an object, GemStone compares the object's authorization attributes in the security policy 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 security policy authorization control access to objects, as shown by Figure 9.1:

Figure 9.1 User Access to Application ObjectSecurityPolicy1



Three users access this application:

- The **System Administrator** owns ObjectSecurityPolicy1 and can read and write the objects assigned to it.
- **User3** belongs to the Personnel group, which authorizes read and write access to ObjectSecurityPolicy1's objects.

- **User2** doesn't belong to a group that can access ObjectSecurityPolicy1, but can still read those objects, because ObjectSecurityPolicy1 gives read authorization to all GemStone users.

Because security policies are objects, access to a GsObjectSecurityPolicy object is controlled by the security policy it is assigned to, exactly like access to any other object. GsObjectSecurityPolicy instances are usually assigned to the DataCuratorObjectSecurityPolicy. The authorization information stored in the GsObjectSecurityPolicy instance, which controls access to the objects assigned to that security policy, does not control access to the policy object itself.

Objects do not "belong" to an security policy. It is more correct to say that objects are associated with a security policy. Although objects know which policy they are assigned to, security policies do not know which objects are assigned to them. Security policies 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, "Resolving Names and Sharing Objects".

9.2 Assigning Objects to Security Policies

For security policy authorizations to have any effect, you must assign some objects to the security policies whose authorizations you have set up.

Default Security Policy and Current Security Policy

In your UserProfile, you may be assigned a *default* security policy, or this may be left empty. When you login to GemStone, your Session uses this default security policy as your current security policy. Any objects you create are assigned to your current security policy; if you do not have a current security policy, the new objects do not have a security policy, and so have world read and write access.

Class UserProfile has the message `defaultObjectSecurityPolicy`, which returns your default GsObjectSecurityPolicy (or nil). Sending the message `currentObjectSecurityPolicy:` to System changes your current security policy:

Example 9.1

```
| aPolicy myPolicy |
myPolicy := System myUserProfile
   defaultObjectSecurityPolicy.
aPolicy := GsObjectSecurityPolicy new.
System commitTransaction.
"change my current security policy to aPolicy"
System currentObjectSecurityPolicy: aPolicy
```

Only committed instances of GsObjectSecurityPolicy can be used.

If you commit after changing the security policy, the new GsObjectSecurityPolicy remains your current security policy until you change the security policy again or log out. If you abort after changing your current security policy, your current security policy is reset from your UserProfile's default security policy.

Unnamed GsObjectSecurityPolicies are often stored in a UserProfile, but named GsObjectSecurityPolicies are stored in symbol dictionaries like other named objects. Private security policies are typically kept in a user's UserGlobals dictionary; security policies for groups of users are typically kept in a shared dictionary.

You can also put security policies in application dictionaries that appear only in the symbol lists of that application's users.

Example 9.2

```
| myPolicy |
"get default security policy"
myPolicy := System myUserProfile defaultObjectSecurityPolicy.
"compare with current"
myPolicy = System currentObjectSecurityPolicy

true
```

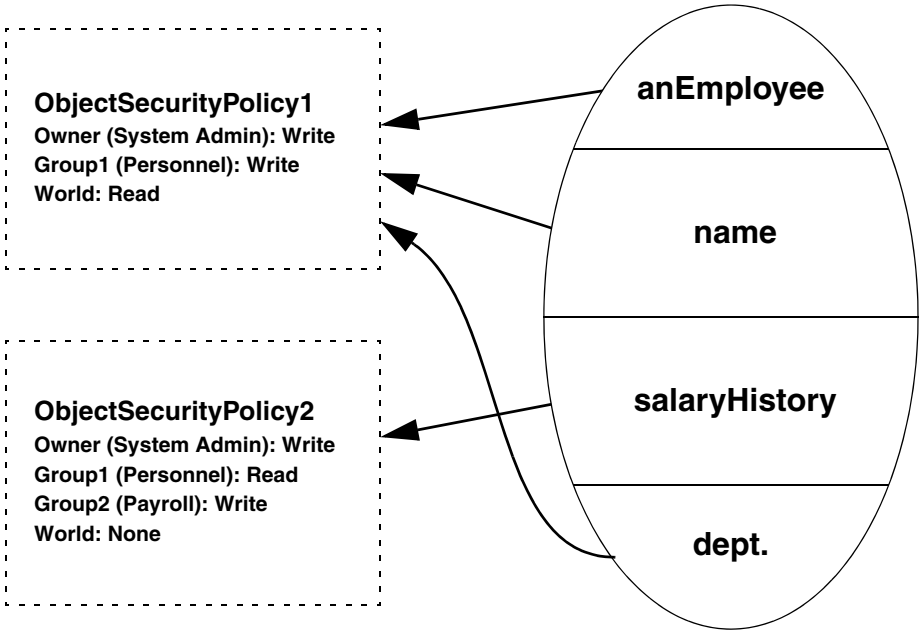
Objects and Security Policies

GemStone object security is defined for objects. 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 security policy assignment. For example, look at anEmployee in Figure 9.2. The contents of its instance variables (name, salary, and department) are separate objects that can be assigned to different security policies. Salary is assigned to ObjectSecurityPolicy2, which enforces more restricted access than ObjectSecurityPolicy1.

Figure 9.2 Multiple Security Policy Assignments for a Compound Object



Collections

When you assign collections of objects to security policies, you must distinguish the container from the items it contains. Each of the items must also be assigned to the proper policy. 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.

Configuring Authorization for an Object Security Policy

Object security polices store authorization information that defines what a particular user or group member can do to the objects with that policy. Three levels of authorization are provided:

write — A user can read and modify any of the objects with that security policy and create new objects associated with the policy.

read — A user can read any of the objects with that security policy, but cannot modify (write) them or add new ones.

none — A user can neither read nor write any of the objects with that security policy.

By assigning a security policy to an object, you give the object the access information associated with that policy. Thus, all objects with a security policy have exactly the same protection; that is, if you can read or write one object with to a certain policy, you can read or write them all.

Controlling authorizations at the security policy 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 security policy 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 security policy. If you are no longer authorized to do so, GemStone issues an error, and your default security policy once again becomes your current security policy. If you are no longer authorized to write in your default security policy, GemStone terminates your session, and you are unable to log back in to GemStone. If this happens, see your system administrator for assistance.

Owner, Group, and World Authorization

A GsObjectSecurityPolicy controls what access a user has to associated objects. Access can be separately assigned for:

- a security policy's *owner*
- *groups* of users (by name)
- the *world* of all GemStone users

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.

These categories overlap. The owner of a security policy is also in the world of all GemStone users, and may also be in one or more groups that have other access authorization. When determining a user's authorization, the most permissive or generous authorization will be allowed and other, more restrictive authorizations, will be ignored. Thus, if world authorization is #read, but the user is a member of a group with #write authorization, then the world authorization will be ignored.

Owner Authorization

Each GsObjectSecurityPolicy has an owner. The owner of a policy may be assigned read, write, or no access in the security policy, and therefore to the objects associated with this security policy. Usually, the owner of a policy has write authorization, but this isn't required (unless this is the default security policy for that user). Users may own more than one security policy.

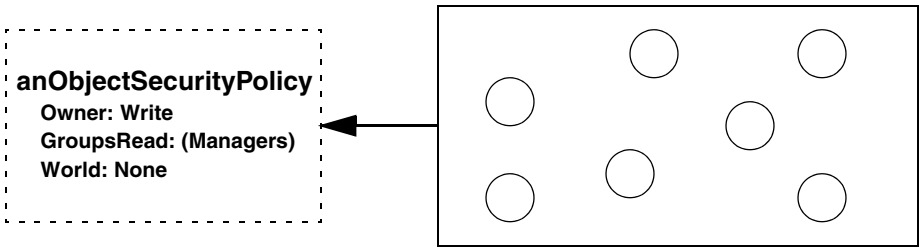
The message `GsObjectSecurityPolicy>>ownerAuthorization:` *anAuthorizationSymbol* is used to set and clear authorization for the owner of the security policy. The message `GsObjectSecurityPolicy>>ownerAuthorization` returns the authorization for the owner of the security policy.

Group Authorization

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 9.3, for example, a group named Managers was set up to allow a few users to read the objects in anObjectSecurityPolicy, while GemStone users in general aren't allowed any access.

Figure 9.3 User Access to a Security Policy's Objects



The global collection AllGroups, a collection of group names, defines all groups in the system. Membership in a group is granted by having adding the group name to the user's UserProfile groups.

The message `GsObjectSecurityPolicy>>authorizationForGroup:` *groupNameString* returns the rights for users in the group *groupNameString*.

The message `GsObjectSecurityPolicy>>groupsWithAuthorization:` *anAuthSymbol* returns the names of groups that have a particular level of access (#read, #write, or #none) for the receiver security policy.

To set group access, use the message `GsObjectSecurityPolicy>>group:` *groupNameString* `authorization:` *anAuthSymbol*. For example, to set the group authorization as shown in Example 9.3, use the following:

```
anObjectSecurityPolicy group: 'Managers' authorization: #read
```

World Authorization

In addition to storing authorization for its owner and for groups, a security policy can also be told to authorize or to deny access by all GemStone users (*the world*.)

The message `GsObjectSecurityPolicy>>worldAuthorization` returns the rights for all users. A corresponding message, `GsObjectSecurityPolicy>>worldAuthorization:` *anAuthSymbol*, sets the authorization for all GemStone users. For example:

```
anObjectSecurityPolicy worldAuthorization: #none
```

Predefined GsObjectSecurityPolicies

The initial GemStone repository has eight GsObjectSecurityPolicies, with the following Ids:

1. **SystemObjectSecurityPolicy**

   This security policy is defined in the Globals dictionary, and is owned by the SystemUser. All GemStone users, represented by world access, are authorized to read, but not write, objects associated with this security policy. The group #System is authorized to write to objects in this policy.

2. **DataCuratorObjectSecurityPolicy**

   This security policy 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 security policy. The group #DataCuratorGroup is authorized to write in this security policy.

   Objects in the DataCuratorObjectSecurityPolicy include the Globals dictionary, the SystemRepository object, all GsObjectSecurityPolicy objects, AllUsers (the set of all GemStone UserProfiles), AllGroups (the collection of groups authorized to read and write objects in GemStone security policies), and each UserProfile object.

   NOTE:
   When GemStone is installed, only the DataCurator is authorized to write in this security policy. To protect the objects in the DataCuratorObjectSecurityPolicy against unauthorized modification, other users should not write in this security policy.

3. **(unnamed)**

   The initial repository does not use this Id. Repositories that have been converted from earlier GemStone/S server products use this for the **GsTimeZoneObjectSecurityPolicy**.

4. **GsIndexingObjectSecurityPolicy**

   This security policy is used by the indexing subsystem.

5. **SecurityDataObjectSecurityPolicy**

   This security policy is used by the system for passwords for UserProfiles, and other highly protected information.

6. **PublishedObjectSecurityPolicy**

   This security policy is used for objects in the Published symbol dictionary.

7. **(unnamed) default GsObjectSecurityPolicy of GcUser**

   This security policy is used by the system for reclaiming storage.

8. **(unnamed) default GsObjectSecurityPolicy of Nameless**

   This security policy is used by Nameless sessions.

For repositories that have been converted from certain earlier versions, there may also be GsObjectSecurityPolicy with id 20, with world write.

Changing the Security Policy for an Object

If you have the authorization, you can change the accessibility of an individual object by assigning a different security policy to it.

The message `Object >> objectSecurityPolicy` returns the security policy that protects that receiver, or nil if the receiver does not have an associated security policy:

Example 9.3

```
UserGlobals objectSecurityPolicy
%
anObjectSecurityPolicy, Number 2 in Repository SystemRepository,
Owner DataCurator write, Group DataCuratorGroup write, World read
```

The message `Object >> objectSecurityPolicy:` *anObjectSecurityPolicy* assigns *anObjectSecurityPolicy* as the security policy for the receiver. You also use this method to remove the security policy, so the receiver object has world read and write access. You must have write authorization for both security policies, that of the receiver and the argument. Assuming the necessary authorization, this example assigns a new security policy to class Employee:

```
Employee objectSecurityPolicy: aPolicy.
```

You may override the method `objectSecurityPolicy:` 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 security policy when you reassign the composite object. You can implement the message `objectSecurityPolicy:` to perform these multiple operations. Within the method `objectSecurityPolicy:` for your composite class, send the message `assignToObjectSecurityPolicy:` to the receiver and each object of which it is composed.

For example, an `objectSecurityPolicy:` method for the class Menagerie might appear as shown in Example 9.4. The object itself is assigned to another security policy using the method `assignToObjectSecurityPolicy:`. Its component objects, the animals themselves, have internal structure (names, habitats, and so on), and therefore call Animal's `objectSecurityPolicy:` method, which in its turn sends the message `assignToObjectSecurityPolicy:` to each component of anAnimal, ensuring that each animal is properly and completely reassigned to the new security policy.

Example 9.4

```
Array subclass: 'Menagerie'
   instVarNames: #()
   classVars: #()
   classInstVars: #()
   poolDictionaries: {}
   inDictionary: UserGlobals
%
method: Menagerie
objectSecurityPolicy: aPolicy
```

```
"Assign the receiver and each component to the given objectSecuri-
tyPolicy."
self assignToObjectSecurityPolicy: aPolicy.
1 to self size do:
   [:eachAnimal | eachAnimal
      objectSecurityPolicy: aPolicy. ]
%
```

Special objects — SmallInteger, SmallDouble, Character, Boolean, and *nil* — are assigned the SystemObjectSecurityPolicy and cannot be assigned another security policy.

Security Policy Ownership

Each GsObjectSecurityPolicy has an owner—by default, the user who created it. An security policy's owner is always has control over who can access the security policy's objects. As a security policy's owner, you can alter your own access rights at any time, even forbidding yourself to read or write objects with that security policy.

You might not be the owner of your default security policy. To find out who owns a security policy, send it the message `owner`. The receiver returns the owner's UserProfile, which you may read, if you have the authorization:

Example 9.5

```
"Return the userId of the owner of the default security policy for
the current Session."
| aUserProf myDefaultPolicy |
"get default security policy"
myDefaultPolicy := System myUserProfile
   defaultObjectSecurityPolicy.
myDefaultPolicy notNil ifTrue:
   ["return its owner's UserProfile"
   aUserProf := myDefaultPolicy owner.
   "request the userId"
   aUserProf userId]
%
user1
```

Every security policy 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 security policy to the user associated with *aUserProfile*:

```
System myUserProfile defaultObjectSecurityPolicy owner: aUserPro-
file
```

In order to reassign ownership of a security policy, you must have write authorization for the DataCuratorObjectSecurityPolicy. Because of the way separate authorizations for owners, groups and world combine, changing access rights for the any one of them may or may not alter a particular user's rights to a security policy.

CAUTION

Do not, under any circumstances, attempt to change the authorization of the SystemObjectSecurityPolicy.

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 security policy or your current security policy. If you lose write authorization for your default security policy, you will not be able to log in again.

Finding Out Which Objects Are Protected by a Security Policy

It may be useful for you to be able to find all the objects that are protected by a particular security policy. An expression of the form:

`SystemRepository listObjectsInObjectSecurityPolicies:` *anArray*

takes as its argument an array of security policy IDs, and returns an array of arrays. Each inner array contains all objects whose security policy ID is equal to the corresponding security policy ID element in the argument *anArray*. Instances to which you lack read authorization are omitted without notification.

Note that this method aborts the current transaction and scans the object header of each object in the repository.

If the result set is very large, there is a risk of out of memory errors. To avoid the need to have the entire result set in memory, the following methods are provided:

`Repository >> listObjectsInObjectSecurityPolicyToHiddenSet:` *anObjectSecurityPolicyId*

This method puts the set of all objects in the specified security policy in the ListInstancesResult hidden set. (a hidden set is an internal memory structure that, while not an object, is treated as one).

To enumerate the hidden set, you can use this method:

`System >> hiddenSetEnumerate:` *hiddenSetId* `limit:` *maxElements*

using a *hiddenSetId* of 1, which is the number of the "ListInstancesResult" hidden set in GemStone/S 64 Bit v3.2. This hidden set number is subject to change in new releases; to determine which hidden sets are in a particular release, use the GemStone Smalltalk method `System Class >> HiddenSetSpecifiers`. For more on hidden sets, see "Other Optimization Hints" on page 277.

You can also list objects that are protected by a particular security policies to an external binary file, which can later be read into a hidden set. To do this, use the method:

`Repository >> listObjectsInObjectSecurityPolicies:` *anArray* `toDirectory:` *aString*

This method scans the repository for the instances protected by the security policies in *anArray* and writes the results to binary bitmap files in the directory specified by *aString*.

Binary bitmap files have an extension of `.bm` and may be loaded into hidden sets using class methods in System.

Bitmap files are named:

`objectSecurityPolicy<`*ObjectSecurityPolicyId*`>-objects.bm`

where *ObjectSecurityPolicyId* is the security policy ID.

The result is an Array of pairs. For each element of the argument *anArray*, the result array contains *ObjectSecurityPolicyId*, *numberOfInstances*. The *numberOfInstances* is the total number written to the output bitmap file.

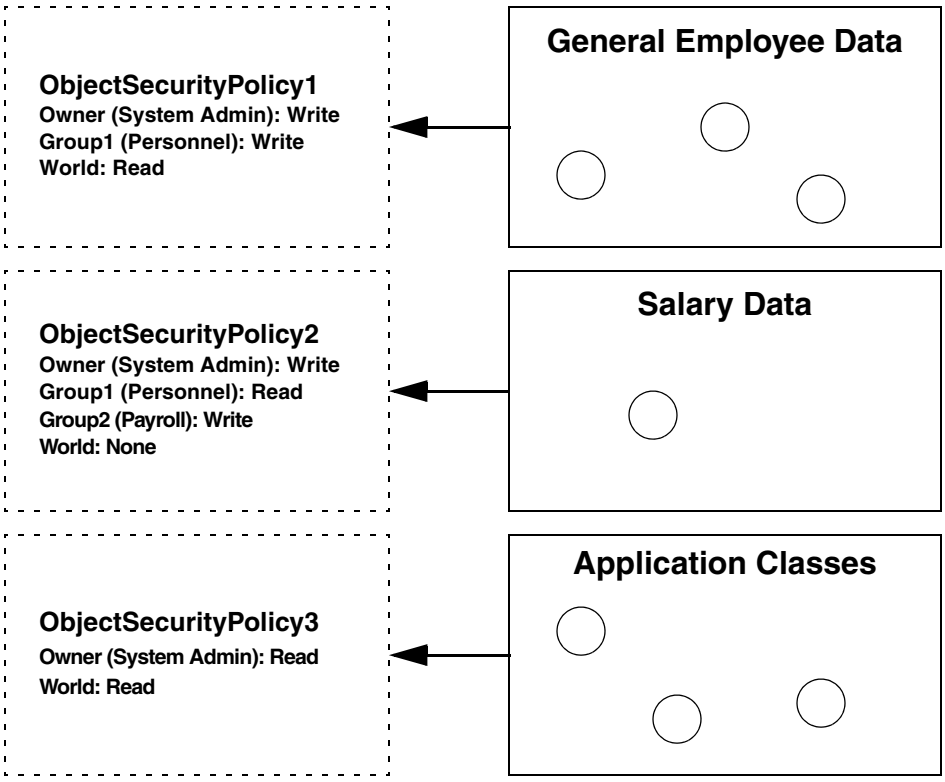
9.3 An Application Example

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 9.4 shows a typical production situation.

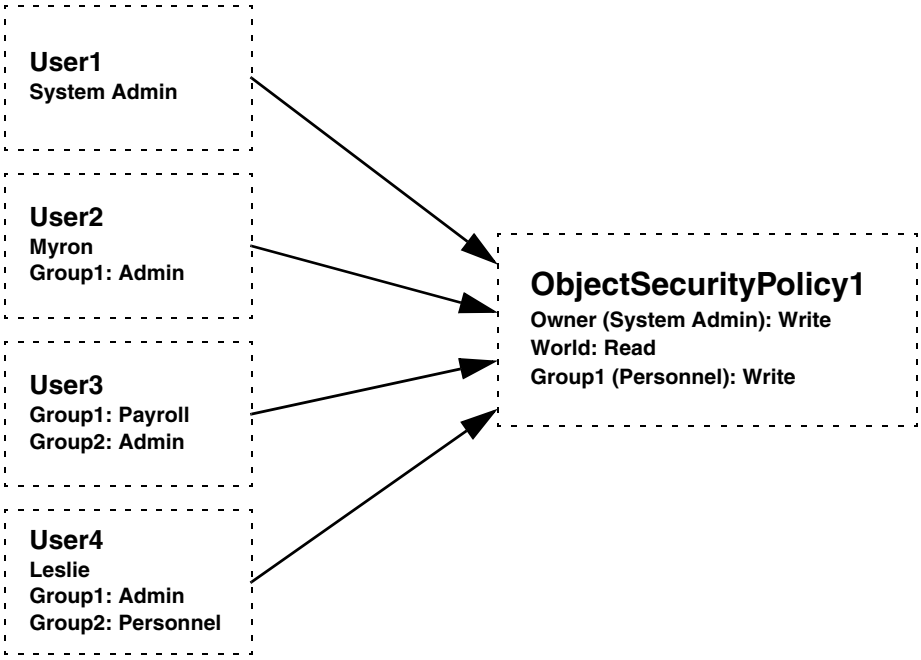
Figure 9.4 Application Objects Assigned to Three Security Policies



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 ObjectSecurityPolicy1, which anyone can look at, but only the Personnel group or owner can change. ObjectSecurityPolicy2 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 ObjectSecurityPolicy3, which no one can change.

Look at how the user name, group membership, and security policy authorization control access to objects, as shown by Figure 9.5 and Figure 9.6:

Figure 9.5 User Access to Application ObjectSecurityPolicy1

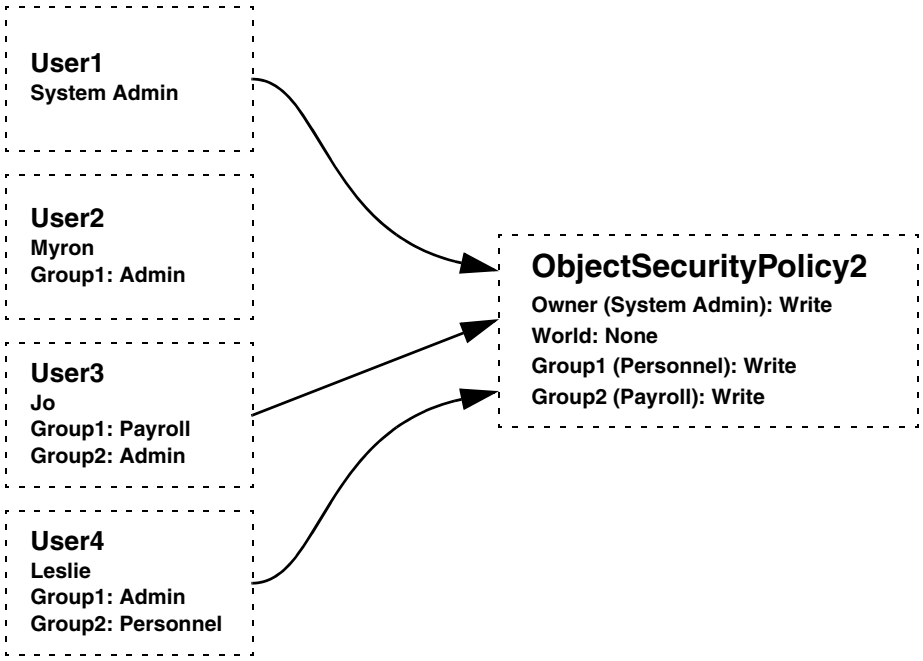


Four users access this application:

- The **System Administrator** owns both security policies and can read and write the objects assigned to them.
- **Leslie** belongs to the Personnel group, which authorizes her to read and write ObjectSecurityPolicy1's objects and read ObjectSecurityPolicy2's objects.
- **Jo** can read and write the objects assigned to ObjectSecurityPolicy2, because she belongs to the Payroll group. She doesn't belong to a group that can access ObjectSecurityPolicy1, but she can still read those objects, because ObjectSecurityPolicy1 gives read authorization to all GemStone users.
- **Myron** does not belong to a group that can access either security policy. He can read the objects assigned to ObjectSecurityPolicy1 objects, because it allows read access to all GemStone users. He has no access at all to ObjectSecurityPolicy2.

Leslie and Jo are sometimes updaters and sometimes reporters, depending on the type of data. Myron is strictly a reporter.

Figure 9.6 User Access to Application ObjectSecurityPolicy2

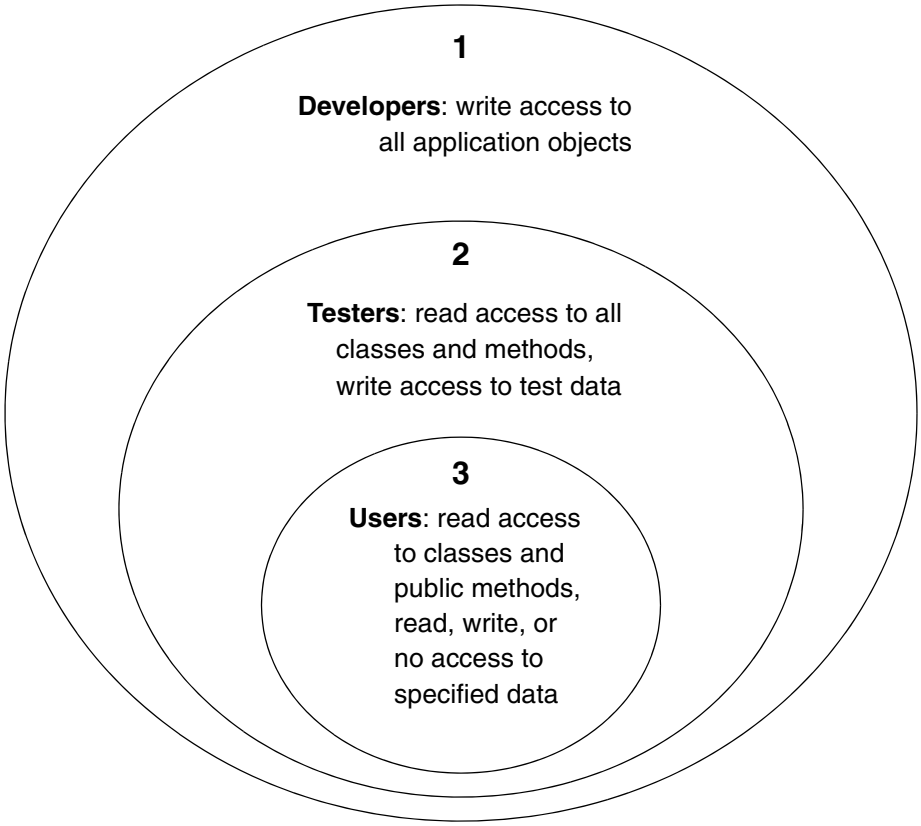


9.4 A Development Example

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 security policies 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 9.7.

Figure 9.7 Access Requirements During an Application's Life Cycle



Planning Security Policies 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. To prevent accidental damage to them, however, you probably want to limit write access. The CodeModification privilege is required to create or modify classes and methods. You can further limit write access using security policies. You may even want to change the owner's authorization to read, until changes are required.

Like other objects, classes and their methods are assigned to security policies on an object-by-object basis. You may keep separate subsections of your application in different security policies, with different write authorizations, if you want.

CodeModification privilege

All application developers will need to have CodeModification privilege. This is in addition to the ability to read and write the appropriate security policies. Without CodeModification privilege, you cannot compile methods or classes, add new methods,

add a Class to a SymbolDictionary, or perform other operations required for application development.

Application users, on the other hand, should not have CodeModification privilege, since they will not be modifying methods or classes. This allows you to protect the application code for inadvertent (or intentional) damage or modification, even if you do not want to implement object level security.

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 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 security policies 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 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, "Resolving Names and Sharing Objects".

Owner

By default, the owner has write access to the objects protected by a security policy. 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 updator. This depends on the type of objects that will be assigned to the security policy.

In Table 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 security policies 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 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 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 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 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 3.

Table 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
vacation-Days	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 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 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
vacation-Days	W	W	W	N	N	N	N

read or write access as Personnel group

Table 4 Access to the Last Six Objects Through the Personnel Group

	Users						
Objects	**System Admin.**	**Human Resource**	**Employ. Records**	**Payroll**	**Mktg**	**Sales**	**Customer Support**
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 3. Because these departments' needs don't match anyone else's, they must each have a separate group.

Table 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 Security Policies

When you have been through this exercise with all your application's prospective objects and users, you are ready to plan the security policies. For easiest maintenance, use the smallest number of security policies that your required combinations of owner, group, and world authorizations allow. You don't need different security policies with duplicate functionality to separate particular objects, like the application classes and data objects. Remember that symbol lists, not security policies, are used to organize objects for listing and retrieval.

In this example you need six security policies, as shown in Figure 9.8. 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 security policies and authorizations you want for your project, you can refer to procedures in the *System Administration Guide* for implementing that plan.

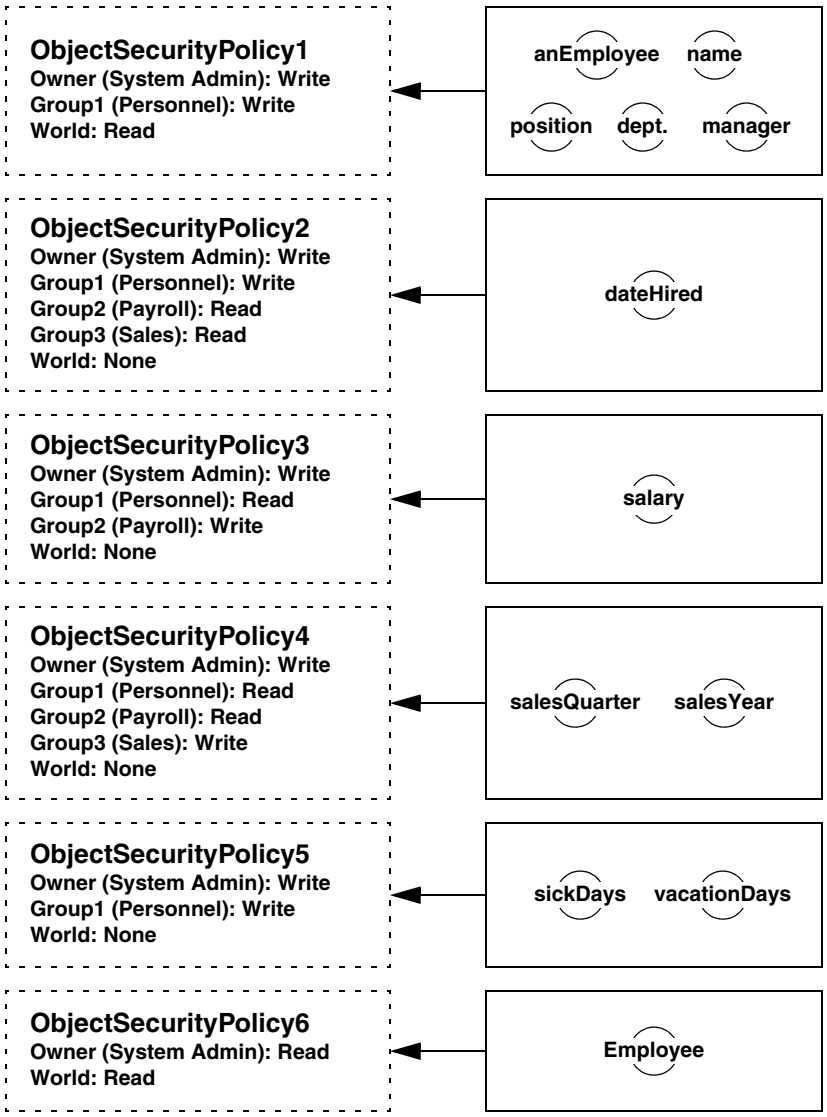
Setting Up Security Policies for Joint Development

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 security policies. 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 security policy assignments in various ways:

- **Full access to all personal security policies.** Give all the developers their own default security policies to work in. Give everyone in the team write access to all the security policies. Because the objects you create are typically assigned to your default security policy, this method may be the simplest way to organize shared work.
- **Read access to all personal security policies.** Set up the same as above, except give everyone read access to the security policies. 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 security policy.** Give all developers the same default security policy, writable by everyone. This is an easy, informal way to share objects.
- **Full access to a shared security policy plus private security policies.** Developers work in their own default security policies and reassign their objects to the shared security policy 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 security policy. The collection has references to many other objects, which can be associated with other security policies. Everyone has the references, but they get errors if they try to access objects with non-readable security policies. 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 9.8 Security Policies 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 security policies.
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 security policies.
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 security policies 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 security policies. 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, "Resolving Names and Sharing Objects."

Security Policy Assignment for User-created Objects

Because security policy 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 security policy 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 Security Policy to the User Account

Set up users with the proper security policy by default. This is a simple way to assure that someone who creates objects in a single security policy doesn't misplace them. To make it impossible to change security policies, rather than just unlikely, you also have to close write access for group and world to all the other security policies.

This solution would work for the Sales and Payroll groups in the example (Figure 9.8 on page 177). They need read access to several security policies, but they only write in one.

The drawback of this solution is that the user can only use one security policy.

Develop the Application to Create the Data Objects

Your best choice is to create objects in the correct security policy, using the `GsObjectSecurityPolicy>>setCurrentWhile:` method. With this method, the application stores data objects in the proper security policies. This provides the most

protection. Besides guaranteeing that the objects end up in the proper security policy, 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.

9.5 Privileged Protocol for Class GsObjectSecurityPolicy

Privileges stand apart from the security policy 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 security policy authorization. Although the owner of a security policy can always use read/write authorization protocol to restrict access to objects protected by a security policy, the DataCurator can override that protection by sending privileged messages to change the authorization scheme.

The following message to GsObjectSecurityPolicy always requires special privileges:

```
new (class method)
newInRepository: (class method)
```

You can always send the following messages to the security policies you own, but you must have special privileges to send them to other security policies:

```
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:

```
'DefaultObjectSecurityPolicy'
'ObjectSecurityPolicyCreation'
'ObjectSecurityPolicyProtection'
```

For a full list of privileges, see the *System Administration Guide* chapter on User Management.

To add security policy creation privileges to your UserProfile, for example, you might do this:

```
System myUserProfile addPrivilege:
   'ObjectSecurityPolicyCreation'.
```

This gives you the ability to execute `GsObjectSecurityPolicy new.`

A similar message, `privileges:`, takes an array of privilege description strings as its argument. The following example adds privileges for security policy creation and password changes:

```
System myUserProfile privileges:
   #('ObjectSecurityPolicyCreation' '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:
   'ObjectSecurityPolicyCreation'
```

Because UserProfile privilege information is typically protected by a security policy that only the data curator can modify, you might not be able to change privileges yourself. You must have write authorization to the DataCuratorObjectSecurityPolicy, or be a member of DataCuratorGroup, in order to do so.

For direction and information about configuring user accounts, adding user accounts and assigning security policies to those accounts, and checking authorization for user accounts, see the *System Administration Guide*.

Chapter

10 Class versions and Instance Migration

Although you 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, even if your design was perfect, real-world changes usually require changes to the schema sooner or later. This chapter discusses the mechanisms GemStone Smalltalk provides 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 GemStone 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.

10.1 Versions of Classes

In order to create instances of a class, the class must be invariant, and invariant classes cannot be modified. While you defined your schema to be as complete as you could at the time you created the classes, inevitably further changes are needed. 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 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 at least one class history, the class history for the class itself. 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.

Defining a New Version

In GemStone Smalltalk classes have *versions*. Each version is a unique and independent 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.

If you define a new class in a SymbolDictionary that already contains an existing class with the same name, it automatically becomes a new version of the previously existing class. This is the most common way of creating new class versions. Instances that predate the creation of the new version remain unchanged, and continue to access the old class's methods, although tools such as GemBuilder or GemTools may provide options to automatically migrate instances to the new class. Instances created after the redefinition have the new class's structure and access to the new class's methods.

When you define a class, the class creation protocol includes an option to specify the existing class of which the new class is a version. See the keyword `newVersionOf:`.

New Versions and Subclasses

When you create a new version of a class—for example, Animal—subclasses of the old version of Animal still point to the old version of Animal as their superclass (unless you are using a tool which provides the option to automatically version and recompile subclasses). If you wish these classes to become subclasses of the new version, you need to recompile the subclass definitions to make new versions of the subclasses, specifying the new version of Animal as their superclass.

One way to do this is to file in the subclasses of Animal after making the new version of Animal (assuming the new version of the superclass has the same name).

New Versions and References in Methods

When you create a new version of a class (such as `Animal`) you typically want your existing code to use the new version rather than the old version. That is, without being recompiled, existing methods containing code like the following should create an instance of the new version rather than of the old version of Animal class:

```
pet := Animal new.
```

As long as the new class version replaces an existing class in the same SymbolDictionary, then references from existing methods will be automatically updated to the new class version.

This works because a compiled method does not directly reference a global (e.g., the class `Animal`), but references a SymbolAssociation in a SymbolDictionary. When you originally compile the method, it resolves the name using an expression similar to the following:

```
System myUserProfile resolveSymbol: #theClassName
```

The compiled method includes the resulting SymbolAssociation, whose key is the name of the global and whose value is the class (or other object). The value can be updated at any time, for example when you create a new version of a class.

This tiny performance penalty is what allows global variables to vary. If you have a global that you know will be constant, then you can reference the value directly from a compiled method by making the SymbolAssociation invariant before compiling the method.

While the SymbolAssociation is updated with the new value by versioning the class within the same SymbolDictionary, keep in mind that under some circumstances you may have a SymbolAssociation that does not reference the latest version, or the version you expect. If you have a newer class with the same name in a different SymbolDictionary, or if you delete and recreate the class, the SymbolAssociation will continue to point to the older class.

Class Variable and Class Instance Variables

When you create a new version of a class, the values in any Class variables or Class Instances variables in the old class are referenced by the new class as well. By default, all versions of a class refer to the same objects referenced from Class or Class instance variables.

10.2 ClassHistory

In GemStone Smalltalk, every class has 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. While they often have the same class name, this is not a requirement; you can rename classes as well as change their structure.

Defining a Class as a new version of an existing Class

When you define a new class in the same symbol dictionary as an existing class with the same name, 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 that is new to a symbol dictionary, the class 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. This is useful if you want to create a version of a class with a different name or in a different symbol dictionary. If the new class version has the same name and is in the same symbol dictionary, it is not necessary to use `newVersionOf:`, since the new class will become a version of the existing class automatically.

For example, suppose your existing class Animal was defined like this:

Example 10.1

```
Object subclass: 'Animal'
   instVarNames: #('habitat' 'name' 'predator')
   classVars: #()
   classInstVars:  #()
   poolDictionaries:  {}
   inDictionary: UserGlobals
```

Example 10.2 creates a class named NewAnimal and specifies that the class shares the class history used by the existing class Animal.

Example 10.2

```
Object subclass: 'NewAnimal'
   instVarNames: #('diet' 'favoriteFood' 'habitat' 'name'
      'predator')
   classVars: #()
   classInstVars:  #()
   poolDictionaries: {}
   inDictionary: UserGlobals
   description: nil
   newVersionOf: Animal
   options: #()
```

If you wish to define a new class Animal with its own unique class history—in other words, the new class Animal is not a version of the old class Animal—you can add it to a different symbol dictionary, and specify the argument *nil* to the keyword `newVersionOf:`. See Example 10.3.

Example 10.3

```
Object subclass: 'Animal'
   instVarNames: #('favoriteFood' 'habitat' 'name'
      'predator')
   classVars: #()
   classInstVars:  #()
   poolDictionaries:  {}
   inDictionary: Published
   description: nil
   newVersionOf: nil
   options: #()
```

If you try to define a new class with the same name as an existing class that you did not create, you will most likely get an error, because you are trying to modify the class history of that class — an object which you are probably not permitted to modify. By specifying a `newVersionOf:` of nil, you can still create this class.

However, we recommend against creating multiple unrelated versions of classes with the same name; this can be confusing and it may be difficult to diagnose problems.

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
```

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; the argument to this message is the class whose history is to be reassigned. For example, suppose that we created NewAnimal using the regular class creation protocol, and did not use the method with the keyword `newVersionOf:`. To later specify that it is a new version of Animal, execute the following expression:

```
Animal addNewVersion: NewAnimal
```

10.3 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. The two classes need not be similarly named, or, indeed, have anything else in common, although it will usually make more sense if they represent the same conceptual object.
- Migration can occur whenever you 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. However, instance that are versions of these older classes will not understand new methods added, and require special consideration if you manage code using fileout, or if you use passivation.
- 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, you should 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.

Moreover, many of the methods discussed below — `allInstances`, `listInstances:`, `migrateInstancesTo:`, and others — abort your current view and thus must be executed in a separate transaction.

After migration succeeds, commit your transaction again before you do any further work. Again, this technique ensures a consistent database from which to proceed.

If you need to migrate many instances of a class, break your work into multiple transactions.

Finding Instances and References

To prepare for instance migration, several 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 classes, and returns an array of sets. Each set contains all instances whose class is equal to the corresponding element in the argument *anArray*.

NOTE

The above method searches the database once for all classes in the array. Executing `allInstances` *for each class would require 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. Each set contains all instances that refer to the corresponding element in the argument *anArray*.

NOTE

Executing either `listInstances:` *or* `listReferences:` *causes an abort. However, if the abort would cause any modifications to persistent objects to be lost, the method will signal a* `TransactionError` *instead.*

Repository-wide scans such as `listInstances:` use a multi-threaded scan that can be tuned to use more or less resources of the system, thereby impacting performance of anything else running on this system to a greater or lesser degree. For details on tuning the multi-threaded scan, see the *System Administration Guide*.

What If the Result Set Is Very Large?

If `Repository>>listInstances:` returns a very large result set, there is a risk of out of memory errors. To avoid the need to have the entire result set in memory, the following methods are provided:

`Repository >> listInstances:` *anArray* `limit:` *aSmallInteger*

This method is similar to `listInstances:`, but returns just the first *aSmallInteger* instances of each of the classes in *anArray*.

`Repository >> listInstancesToHiddenSet:` *aClass*

This method puts the set of all instances of *aClass* in a new hidden set (an internal memory structure that, while not an object, is treated as one).

To enumerate the hidden set, you can use this method:

`System Class >> hiddenSetEnumerate:` *hiddenSetId* `limit:` *maxElements*

using a *hiddenSetId* of 1, which is the number of the "ListInstancesResult" hidden set in GemStone/S 64 Bit v3.2. This is the hidden set in which listInstances results are placed. This hidden set number is subject to change in new releases. To determine which hidden sets are in a particular release, use the GemStone Smalltalk method `System Class >> HiddenSetSpecifiers`.

For more on how to use hidden sets, see the section "Other Optimization Hints" on page 277.

You can also list instances to an external binary file, which can later be read into a hidden set. To do this, use the method:

`Repository >> listInstances:` *anArray* `toDirectory:` *aString*

This method scans the repository for the instances of classes in *anArray* and writes the results to binary bitmap files in the directory specified by *aString*. Binary bitmap files have an extension of `.bm` and may be loaded into hidden sets using class methods in System.

Bitmap files are named:

className`-`*classOop*`-instances.bm`

where *className* is the name of the class and *classOop* is the object ID of the class.

The result is an Array of pairs. For each element of the argument *anArray*, the result array contains *aClass*, *numberOfInstances*. The *numberOfInstances* is the total number written to the output bitmap file.

List Instances in Page Order

For even more efficient migration of large sets of objects of multiple classes, you can list all the instances of all the classes in page order - the same order as the objects are stored on disk. This allows multiple objects of several different classes on the same page in the repository to be migrated at the same time.

If migration performance is an issue for your application, the following methods can be used to write the list of instances to a file, and open, read, and process the instances from the file.

```
Repository >> listInstancesInPageOrder: anArray toFile: aString
Repository >> openPageOrderOopFile: aString.
Repository >> readObjectsFromFileWithId: fileId
   startingAt: startIndex upTo: endIndex into: anArray.
Repository >> closePageOrderOopFileWithId: fileId
Repository >> auditPageOrderOopFileWithId: fileId
```

For details on these methods and how to use them, refer to the method comments in the image.

Since the normal operation of the repository, where objects are added, removed, and modified, will cause objects to move from page to page, over time the actual ordering of the objects by page will diverge from the order of the results. When the file is read later, it will (of course) not contain any references to objects that were created since the listInstances was run. During the read, if any of the instances have been garbage collected, the Array of results will contain a nil. Given these issues, it is important to read and process the file as soon as possible after it is created.

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 10.4

```
| 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.

Neither of these messages changes 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 that has been defined 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. The following example migrates the specified instances of Animal to instances of NewAnimal:

```
Animal migrateInstances: #{aDugong . aLemming} to: NewAnimal.
```

Alternatively, the message `migrateInstancesTo:` migrates *all* instances of the receiver to the specified destination class. The following example migrates all instances of Animal to instances of NewAnimal:

```
Animal migrateInstancesTo: NewAnimal.
```

NOTE

Executing either `migrateInstances:to:` *or* `migrateInstancesTo:` *causes an abort. To avoid loss of work, always commit your transaction before you begin data migration.*

Example 10.5 uses `migrateInstances:to:` to migrate all instances of all versions of a class, except the latest version, to the latest version.

Example 10.5

```
| 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: Animal currentVersion].
```

The migration methods `migrateInstancesTo:` and `migrateInstances:to:` return an array of four collections. The first two collections in the array are always empty.

- The third collection is a set of objects that are instances of indexed collections, and were not migrated. See the following discussion, "Migration Errors".
- The fourth collection is a set of objects whose class was not identical to the receiver—presumably, incorrectly gathered instances—and thus, were not migrated. See "Instance Variable Mappings" on page 191.

If all four of these collections are empty, all requested migrations have occurred.

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.

Migrating self

Sometimes a requested migration operation can cause the interpreter to halt and display an error message of the following form:

```
The object <anObject> is present on the GemStone Smalltalk
stack, and cannot participate in a become.
```

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, rewrite the method that sends the migration message to *self*, so as to accomplish its purpose in some other manner.

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. Modify the indexing code as appropriate for the new class version and re-create the index.
  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.

For more information about indexing, see Chapter 7, “Indexes and Querying.”

For more information about committing and aborting transactions, see Chapter 8, "Transactions and Concurrency Control."

Instance Variable Mappings

Earlier, we explained that migration can involve changing the structure of an object. Since migration is only useful if you can retain the data that is contained in these instances, you can set up mappings so instances using the old structure can be migrated to a new structure and updated appropriately.

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 in Example 10.6.

Example 10.6

```
| 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 10.7 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 10.7

```
| 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

This section describes two kinds of customization:

- To initialize an instance variable with the value of a variable that has a different name, you must provide an explicit mapping from the instance variable names of the older class to the instance variable names of the migration destination.

- 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 can implement methods to transform the variable values.

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, it is impossible to determine the value to which *species* ought to be initialized. 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 as shown in Example 10.8.

Example 10.8

```
instVarMappingTo: anotherClass
| result myNames itsNames dietIndex |
"Use the default strategy first to properly fill in inst vars hav-
ing the same name."
result := super instVarMappingTo: anotherClass.
myNames := self allInstVarNames.
itsNames := anotherClass allInstVarNames.
dietIndex := myNames indexOfValue: #diet.
dietIndex > 0
   ifTrue: [(result at: dietIndex) = 0
   ifTrue:[ result at: dietIndex
      put:(itsNames indexOfValue: #favoriteFood)]].
^result
```

The method `allInstVarNames` is used because it would also migrate all inherited instance variables, although at the expense of performance. If your class inherits no instance variables, you could use the method `instVarNames` instead, for efficiency.

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 10.9

```
Object subclass: #OldPoint
   instVarNames: #( #x #y )
   classVars: #()
   classInstVars: #()
   poolDictionaries: {}
   inDictionary: UserGlobals.

oldPoint compileAccessingMethodsFor: OldPoint instVarNames.

Object subclass: #Point
   instVarNames: #( #radius #angle )
   classVars: #()
   classInstVars: #()
   poolDictionaries: {}
   inDictionary: UserGlobals.

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 := (y/x) asFloat arcTan.
   ^self

Point new migrateFrom: (OldPoint new x: 123; y: 456)
   instVarMap: 'unused argument'.
a Point
  radius            4.7229757568719322E02
  angle             2.6346654103491746E-01
```

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 modify the above method as follows:

Example 10.10

```
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 := (y/x) asFloat arcTan.
   ^self]
    ifFalse: [^super migrateFrom: oldPoint instVarMap: aMap]
```

Chapter

11 File I/O and Operating System Access

As a GemStone application programmer, you'll seldom need to be concerned with the details of operating system file management. However, it can be useful 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, as well as other tasks involving operating system access.

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
how to execute operating system commands from GemStone.

File In and File Out
filing out your application source code.

PassiveObject
describes the mechanism that GemStone provides for storing the objects that represent your data.

Creating and Using Sockets
describes the protocol provided by class GsSocket and GsSecureSocket to create operating system sockets and exchange data between two independent interface processes.

11.1 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 comment for the class GsFile in the image.

Instances of GsFile understand most protocol common to Streams.

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. Wildcard 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. (This may not necessarily be the same machine on which the Stone 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, the way in which 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 GemStone 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 want to worry about these details, supply full path names and avoid the use of environment variables. This allows your application to work uniformly across different environments.

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 GemStone Smalltalk using several class methods for GsFile. Example 11.1 creates a file named aFileName in the current directory on the client machine.

Example 11.1

```
| myFile myFilePath |
myFilePath := 'aFileName'.
myFile := GsFile openWrite: myFilePath.
"Here would go code to write data to the file"
myFile close
%
```

Example 11.2 creates a file named aFileName in the current directory on the server.

Example 11.2

```
myFile := GsFile openWriteOnServer: mySpec
"Here would go code to write data to the file"
myFile close
%
```

These methods return the instance of GsFile that was created, or nil if an error occurred. Common errors include invalid paths or insufficient permissions. To determine the specific problem, use the techniques described under "GsFile Errors" on page 204.

Opening a File

GsFile provides a wide variety of protocol to open files. For a complete set of methods, see the image. These methods return the GsFile instance if successful, or nil if an error occurs.

Table 1 GsFile Class Methods to Open Files

Method	Description
`openRead:` `openReadCompressed:`	Opens a file on the client machine for reading, replacing the existing contents.
`openWrite:` `openWriteCompressed:`	Opens a file on the client machine for writing. Creates a new file if one does not exist, or truncates an existing file to 0.
`openAppend:`	Opens a file on the client machine for writing, appending the new contents instead of replacing the existing contents. Creates the file if it does not exist.
`openReadOnServer:` `openReadOnServerCompressed:`	Opens a file on the server for reading, replacing the existing contents.
`openWriteOnServer:` `openWriteOnServerCompressed:`	Opens a file on the server for writing. Creates a new file if one does not exist, or truncates an existing file to 0.
`openAppendOnServer:`	Opens a file on the server for reading, appending the new contents instead of replacing the existing contents.
`GsFile close`	Closes the receiver.
`closeAll`	Closes all open GsFile instances on the client machine except stdin, stdout, and stderr.
`closeAllOnServer`	Closes all open GsFile instances on the server except stdin, stdout, and stderr.

Closing a File or Files

The following methods close the current instance, or multiple open files:

Table 2 GsFile Method Summary

Method	Description
`GsFile >> close`	Closes the receiver.
`GsFile class >> closeAll`	Closes all open GsFile instances on the client machine except stdin, stdout, and stderr.
`GsFile class >> closeAllOnServer`	Closes all open GsFile instances on the server except stdin, stdout, and stderr.

Your operating system limits the number of files a process can concurrently access. 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:` 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 11.1

```
myFile := GsFile openWrite: 'myFileName'.
myFile nextPutAll: 'All of us are in the gutter,'.
myFile cr.
myFile nextPutAll: 'but some of us are looking at the stars.'.
myFile close.
myFile := GsFile openRead: 'myFileName'.
myFile contents.
%

GsFile closeAll.
%
```

These methods return the number of bytes that were written to the file, or nil if an error occurs.

Writing Extended Characters To a File

Characters outside the base ASCII range of 0...255 require multiple bytes to represent. Instance of traditional and Unicode Strings handle characters with larger values than can be held by the given class, by transparently morphing into a class that can hold larger Characters. This is discussed in Chapter 5.

Most GsFile protocol can read and write byte values only, and do not handle Characters with values larger than 255, nor instances of DoubleByteString, QuadByteString, Unicode16 and Unicode32 that include Characters over 255.

To write a string containing extended characters to a file, it must be explicitly encoded into UTF-8 before writing to the file. You can do this by using

```
GsFile >> nextPutAsUtf8: aString
```

This writes the given string (or instsance of Uft8) to the file encoded as UTF-8.

You can also encode a string into UTF-8 using `String >> encodeAsUTF8`. This creates an instance of Uft8, a type of ByteArray holding the UTF-8 encoded equivalent of the data. This can be written to the GsFile normally.

For example, the Euro character € has the Unicode value U+20AC.

Example 11.2 Writing the Extended Character € to a File

```
| myfile str |
myfile := GsFile openWrite: 'extendedCharacterExample.txt'.
str := String new.
str add: 'How to write a Euro character '.
str add: (Character withValue: 16r20AC).
str add: ' to a file'; lf.
myfile nextPutAsUtf8: str.
myfile close.
```

To read a file containing data encoded in UTF-8, use the method

```
GsFile >> contentsAsUtf8
```

This returns the contents of the entire file as an instance of Utf8, which contains byte values. The method `Utf8 >> decodeFromUTF8` can them be used to decode the contents into an instance of the appropriate Unicode String class.

Note that when reading files containing Characters with codePoints larger than 127. you must be aware of the whether the file is encoded in order to decode appropriately. GsFile reads the bytes and does not distinguish between encoded or un-encoded contents.

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 GemStone Smalltalk. For example, the `contents` method (at the end of Example 11.1) returns the entire contents of the specified file and positions the pointer at the end of the file.

In Example 11.3, `next: into:` takes the 12 characters after the current pointer position and places them into the specified string object. It then advances the pointer by 12 characters.

Example 11.3

```
| result |
result := String new.
myFile := GsFile openRead: 'myFileName'.
myFile next: 12 into: result.
myFile close
result.
%
```

Positioning

You can also reposition the pointer without reading characters, or peek at characters without repositioning the pointer. The method:

```
GsFile peek
```

allows you to view the next character in the file without advancing the pointer.

To advance the pointer without reading the intervening characters, use:

```
GsFile skip: anInteger
```

Testing Files

The class GsFile provides a variety of methods that allow you to determine facts about a file.

To test for existence of a file, use:

```
GsFile >> exists: aFileNameString
GsFile >> existsOnServer: aFileNameString
```

These methods returns true if the file exists, false if it does not, and nil if an error occurred.

Renaming Files

Files on the client or server can be renamed or moved. For example:

```
GsFile rename: '/tmp/myfile.txt' to: '/tmp/newname.txt'.

GsFile renameFileOnServer: '$GEMSTONE/data/system.conf' to:
'/users/david/mysystem.conf'.
```

Removing Files

To remove a file from the client machine, use an expression of the form:

```
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 11.4

```
GsFile contentsOfDirectory: '$GEMSTONE/examples/admin' onClient:
false
%
an Array
  #1 /dbf/gsadmin/GS6432/examples/admin/.
  #2 /dbf/gsadmin/GS6432/examples/admin/..
  #3 /dbf/gsadmin/GS6432/examples/admin/onlinebackup.sh
  #4 /dbf/gsadmin/GS6432/examples/admin/archivelogs.sh
```

If the argument is a directory name, this message returns the full pathnames of all files in the directory, as shown in Example 11.4. However, if the argument is a filename, this message returns the full pathnames of all files in the current directory that match the filename. The argument can contain wildcard characters such as *. The following example shows a different use of this message.

```
GsFile contentsOfDirectory: '$GEMSTONE/ver*' onClient: false
%
an Array
  #1 /dbf/gsadmin/GS6432/version.txt
```

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:

All the above methods, like most GsFile methods, return nil if an error occurs.

GsFile Errors

GsFile operations return nil in cases where an error occurs during the operation. For this reason, most GsFile operations should check for nil return. There are separate methods to check for errors within file operations on server files and client files.

To check for errors in an operation on a server file, the method is `GsFile >> serverErrorString`. It is nil if no error has occurred. This error is available until the next GsFile operation is executed.

Example 11.5

```
| myFile |
myFile := GsFile openReadOnServer: 'nonexistentfile'.
myFile isNil
  ifTrue: [GsFile serverErrorString]
  ifFalse: ['Succesfully opened'].
%
errno=2,ENOENT, The file or directory specified cannot be found
```

To check for similar errors for a client file, use the method `lastErrorString`. For example:

Example 11.6

```
| myFile |
myFile := GsFile openRead: 'privatefile'.
myFile isNil
  ifTrue: [GsFile lastErrorString]
  ifFalse: ['Succesfully opened'].
%
errno=13,EACCES, Authorization failure (permission denied)
```

11.2 Executing Operating System Commands

Simple 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 GemStone Smalltalk string. For example:

```
System performOnServer: 'date'
%
Mon Mar 10 15:19:56 PDT 2014
```

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.

The GemStone (reverse) privilege NoPerformOnServer controls the ability to execute this method. If a user account is given this privilege, that user cannot execute `performOnServer:`.

More complex interactions

`System >> performOnServer:` can execute arbitrary OS code on the server, but only operates synchronously; Smalltalk will block until the command has completed.

To provide an asynchronous perform, and to allow Smalltalk to read from stdout or write to stdin, you can use the class GsHostProcess.

To use this, use the class method `fork:`, passing the command line you wish to execute. This will return immediately with an instance of GsHostProcess with sockets on stdin, stdout, and stderr. You can use socket protocol to read from or write to these sockets.

Note that pathname resolution is not provided. You must fully qualify executable paths.

For example:

```
run
| hostprocess |
hostprocess := GsHostProcess fork: '/bin/date'.
hostprocess stdout read: 1024
%
Tue Mar 11 11:03:14 PDT 2014
```

11.3 File In and File Out

To archive your application or transfer GemStone classes to another repository you can *file out* GemStone 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.

Fileout

Methods in behavior allow you to file out a class, category, or method. For example, to file out a single class named Customer:

```
| myFile |
myFile := GsFile openWrite: 'CustomerClassFileout.gs'.
myFile isNil
  ifTrue: [^GsFile serverErrorString].
Customer fileOutClassOn: myFile.
myFile close.
%
```

Using ClassOrganizer, you can file out all the classes and methods in a SymbolDictionary, ordered correctly for filein. For example, to file out UserGlobals:

```
| myFile |
myFile := GsFile openWrite: 'UserGlobalsFileout.gs'.
myFile isNil
  ifTrue: [^GsFile serverErrorString].
ClassOrganizer new fileOutClassesAndMethodsInDictionary:
   UserGlobals on: myFile.
myFile close.
%
```

File out can also be done using the topaz command **fileout**. See the *Topaz User's Guide* for more information.

Filein

File in is done using topaz **input** command, or facilities provided by GBS.

For example, to file in the fileout of UserGlobals from the previous example:

```
topaz 1> input UserGlobalsFileout.gs
```

Handling strings with extended characters

GsFile cannot directly write characters with codepoints over 255. If your class or method names, code, or comments includes any characters with codepoints over 255, you will need to encode as UTF-8 in order to file out using GsFile, as is described on "Writing Extended Characters To a File" on page 201.

In topaz, the **fileformat** command allows you to file out and file in as UTF-8.

11.4 PassiveObject

To archive your data, you can *passivate* objects themselves to a file. Objects representing your data are stored into a serialized, text-based form by the GemStone class PassiveObject. PassiveObject starts with a root object and traces through its instance variables, and their instance variables, recursively until it reaches special objects (instances of SmallInteger, Character, Boolean, SmallDouble, or UndefinedObject), or classes that can be reduced to special 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.
- SystemRepository cannot be preserved.
- Blocks that refer to globals or other variables outside the scope of the block cannot be reactivated correctly.
- 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.
- Identities (OOPs) are not preserved.

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.

In addition, since object identity is not preserved, behavior that depends on identity may not work as expected. For example, for objects that implement = using ==, the re-activated object will not be = to the original.

The class PassiveObject implements the method `passivate:` *anObject* `toStream:` *aGsFileOrStream* to write objects out to a stream or a file. To write the object AllEmployees out to the file *allEmployees.obj* in the current directory, execute an expression of the form shown in Example 11.7.

Example 11.7

```
| empFile |
empFile := GsFile openWriteOnServer: 'allEmployees.obj'.
PassiveObject passivate: AllEmployees 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 11.8

```
| empFile passivatedEmployees |
empFile := GsFile openReadOnServer: 'allEmployees.obj'.
passivatedEmployees := PassiveObject newOnStream: empFile.
AllEmployees := passivatedEmployees activate.
empFile close.
```

Examples 11.7 and 11.8 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.

11.5 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 GemStone 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 addition to standard sockets created by GsSocket, you can create secure SSL sockets using the class GsSecureSocket. GsSecureSocket is a subclass of GsSocket that adds protocol to specify certificates and require authentication.

Both GsSocket and GsSecureSocket contain class methods `clientExample` and `serverExample`, and GsSecureSocket contains additional class methods `clientExample2` and `serverExample2`. These methods provide examples of how to create a socket connection between two sessions. 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 execute these methods from Topaz or from GemBuilder for Smalltalk, but note that `serverExample`, which should be started first, will take control of the interface until the `clientExample` completes the socket connection.

GsSocket

GsSocket is the class representing a basic socket.

Establishing the connection

To setup a socket connection, you create instances of GsSocket in both the client and server processes.

1. On the server side, create an instance of GsSocket, and call `makeServerAtPort:` This creates a listening socket on the given port.

   To have the operating system select a port, use a wildcard bind using `makeServer:`, or pass nil as the port argument. You will then need to determine the port that the client should connect at using the `port` method.

2. On the client side, create an instance of a GsSocket and call one of the following:

   - `connectTo:` for a connection to a process on the same host
   - `connectTo:on:` if the server is on a different machine
   - `connectTo:on:timeoutMs:` to specify a timeout for the connection

   Provided there was a listening server socket setup as in step 1, this will initiate the connection to the server.

3. The server then does an `accept`, or `acceptTimeoutMs:` (to specify a timeout) . This returns a new instance of GsSocket for the client connection.

   Note that the server side has two sockets; a listening socket and the established socket with the client.

Communication on the socket

Each process can write and read to the socket using protocol such as `write:` and `read:`. See the image methods in the categories Reading and Writing for specific methods.

Writes and reads are of byte objects such as String or ByteArray. Read operations are for a specified number of bytes, and return the actual number of bytes read if fewer bytes were available (if fewer bytes were written to the socket by the peer).

Closing the socket

When completed, the client should close its socket and the server close the listening and established sockets. This is done by simply sending `close` to the sockets.

Socket Configuration

Socket configuration can be done using the method

```
GsSocket >> option:put:
```

See the comments in this method for details on socket configuration.

The most common option is blocking.

Blocking

Sockets can be made blocking or non-blocking, and the blocking status checked, using the following methods:

```
GsSocket >> makeNonBlocking
GsSocket >> makeBlocking
GsSocket >> isNonBlocking
GsSocket >> isBlocking
```

GsSecureSocket

GsSecureSocket creates a secure socket using Secure Sockets Layer (SSL), providing access to the open-source OpenSSL library. This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit. (http://www.openssl.org/)

To create a secure socket, you create instances of the GsSecureSocket class and first establish the connection as a regular socket. Then, further protocol authenticates the connection to make the socket secure.

Secure Sockets include class level setup of certificates and types of authentication that may be done outside of specific socket operations.

Set up certificates and private keys

GsSecureSocket instances must be configured with the CA certificates, private key files, and passphrases (if needed), to allow them to complete the secure handshake.

These can be configured in the class GsSecureSocket, so they will apply to all instances of GsSecureSocket. In this case they must be configured before the instance of GsSecureSocket is created.

Alternatively, you can create the instance of GsSecureSocket, and send instance methods to configure the certificates and private keys.

Generating certificates

To use secure sockets, you will need to have certificates, CA certificates, private key files and passphrases, such as sufficient for your security requirements. These may be provided by your organization.

The GemStone distribution includes example certificates, and a script that will allow you to generate certificates. The script is provided here:

```
$GEMSTONE/examples/openssl/make_example_certs.sh
```

The GemStone distribution also includes the openssl executable:

```
$GEMSTONE/bin/openssl
```

This is the version of openssl that GemStone uses; using this, rather than any version that may be present on your system, is recommended. For details on the openssl interface, see

```
http://www.openssl.org/docs/apps/openssl.html
```

Enable or disable certificate validation

By default, client sockets do not validate server connections, but by default server sockets validate client connection. This is the usual case, and if this is your preferred behavior you do not need to explicitly enable or disable validation.

If validation on either server socket or client socket is disabled, the methods that configure the certificates, CA certificates, private key files and passphrases do not need to be executed.

Class methods that configure validation must be executed prior to the creation of the GsSecureSocket instance.

Server Sockets

By default, server sockets validate all connection requests. To disable validation, use the methods:

```
GsSecureSocket enableCertificateVerificationOnServer
aGsSecureServerSocket enableCertificateVerification
```

Parallel methods exist to re-enable validation after validation is disabled.

Client Sockets

By default, client sockets do not validate connections from the server. To enable validation, use the methods:

```
GsSecureSocket enableCertificateVerificationOnClient
aGsSecureClientSocket enableCertificateVerification
```

Parallel methods exist to disable validation after validation is enabled.

Setup CA certificates

When socket validation is to be done, the Certificate Authority (CA) certificates should be setup prior to creating instance of GsSecureSocket.

Server Sockets

By default, server sockets validate client connections. Before creating a server socket, the CA certificate used to validate client connections should be loaded using the following method:.

```
GsSecureSocket class >>
   useCACertificateFileForServers: certfile
```

The CA certificate file must be in PEM format. It may contain more than one certificate. If this method returns false, an error occurred and the certificate was not successfully loaded.

An example CA certificate file is provided here:

```
$GEMSTONE/examples/openssl/certs/serverCA.pem'
```

Client Sockets

By default, certificates are not verified on the client, so the client CA certificate file does not need to be loaded. If you need to enable client validation, load the client CA certificate file using the following method.

```
GsSecureSocket class >>
   useCACertificateFileForClients: certfile
```

Setup certificate, private key, and passphrase

The certificate, private key, and private key passphrase can be setup by class methods to apply to all instances, or by sending messages to the instance of GsSecureSocket.

Method variants are provided that allow you to pass in the certificate and private key either as file path and name, or as a string. If a string is used, it must exactly match the contents of the corresponding certificate file (including white space, line feeds, etc.), or the strings will not be accepted.

Both certificate and private key must be in PEM format, and the private key must match the certificate. The same file may be specified for the certificate file and the private key file.

The certificate may contain a certificate chain or a single certificate.

If the private key requires a passphrase, it must be specified as a string. If the private key does not require a passphrase, the argument is expected to be nil.

Class setup for server sockets

To specify the server certificates, private key file, and passphrase (if required), that will be used for all secure server sockets that are created after these methods are invoked, use the methods:

```
GsSecureSocket class >> useServerCertificateFile: certfile withPri-
vateKeyFile: keyFile privateKeyPassphraseFile: strOrNil

GsSecureSocket class >> useServerCertificate: certString withPri-
vateKey: keyString privateKeyPassphraseFile: strOrNil
```

An example file is provided here:

```
$GEMSTONE/examples/openssl/certs/server.pem
```

Class setup for client sockets

By default, certificates are not verified on the client, so the certificate and private key do not need to be setup.

If you need to enable client validation, the follow methods specify the client certificates, private key file, and passphrase, that will be used to validate server connections for secure client sockets that are created after these methods are invoked:

```
GsSecureSocket class >> useClientCertificateFile: certfile withPri-
vateKeyFile: keyFile privateKeyPassphraseFile: strOrNil

GsSecureSocket class >> useClientCertificate: certString withPri-
vateKey: keyString privateKeyPassphraseFile: strOrNil
```

Instance setup for client or server sockets

You can specify the certificate, private key, and passphrase for a single specific instance of GsSecureSocket (either a server socket or a client socket), using instance methods:

```
GsSecureSocket >> useCertificateFile: certfile withPrivateKeyFile:
keyFile privateKeyPassphraseFile: strOrNil
GsSecureSocket >> useCertificate: certString withPrivateKey: keyString
privateKeyPassphraseFile: strOrNil
```

Setup the Cipher list

The list of ciphers that are acceptable to use can be configured, either on the class side for servers and clients, or for specific instances of GsSecureSocket client or server sockets.

The cipher list is specified as a formatted string. See http://www.openssl.org/docs/apps/ciphers.html for details on the format of this string (as well as other information on ciphers).

For example, to use all ciphers except NULL ciphers and anonymous Diffie-Hellman (DH), and sort by strength, use the following string:

```
'ALL:!ADH:@STRENGTH'
```

To configure the cipher list for all instances of GsSecureSocket, use the following methods. These methods return true if the specification finds one or more usable ciphers, false if no usable ciphers match the specification.

```
GsSecureSocket class >>
   setClientCipherListFromString: aString
GsSecureSocket class >>
   setServerCipherListFromString: aString
```

To configure the cipher list for a specific instance of a server socket or client socket, the ciphers must be set before `secureConnect:/secureAccept` are executed. This method returns true if the specification finds one or more usable ciphers, false if no usable ciphers match the specification, and nil if the operation has no affect because the receiver is already connected.

```
GsSecureSocket class >>
   setCipherListFromString: aString
```

Once an instance of GsSecureSocket is successfully connected, you can fetch the cipher in use using:

```
GsSecureSocket >> fetchCipherDescription
```

Establishing the connection

Rather than creating instances using `GsSocket class >> new`, with GsSecureSocket sockets are instantiated using `newClient` and `newServer`.

To establish the socket connection, as with regular GsSocket,

1. The server creates the socket using `newServer`, and calls `makeServerAtPort:` on an unused port, to create the server listener socket on that port.
2. The client creates the socket using `newClient`, and calls `connectTo:`, specify the same port as in Step 1.
3. The server socket calls `accept`, which creates the connected socket on the given port.

This establishes the standard socket, but the connection is not secure. Another client-server interaction is required to make this a secure socket.

At this point, you can setup specific certificates and ciphers that will apply to these sockets only, as described in the preceding sections. This is needed if you have not previously set up certificates and ciphers that apply to all GsSecureSocket connections.

Then continue with the process that makes the socket secure:

4. The client socket calls `secureConnect`
5. The server socket calls `secureAccept`

If these methods return true, then the connection is secure. To determine if you have a secure connection, use the method:

```
GsSecureSocket >> hasSecureConnection
```

Communication on the socket

At this point reads and writes are done as for standard sockets.

Closing the socket

You can either close the socket connection entirely, or close the secure connection and remain connected for normal (not secure) communication.

To close the socket entirely, use

```
GsSecureSocket >> close
```

Which performs both the secure close and the regular close.

Note that the secure `close` requires a handshake. If the socket is blocking, and the peer does not respond, then the close will hang. To close the socket, we recommend first making it non blocking:

```
mySecureSocket makeNonBlocking.
mySecureSocket close.
```

To close only the secure socket and leave the connection available for non-secure communication, you can use the method

```
GsSecureSocket >> secureClose
```

Which must be executed by both sockets on the connection. You can then call `close` later, to close the connection entirely.

Error handling

GsSocket

The following methods are implemented both for the class and instance of GsSocket. For errors in GsSocket class methods, use the class side error methods, and for errors in GsSocket instance methods, use the instance methods

`lastErrorString`
: Returns a String containing information about the last error or nil if no error has occurred. Clears the error information.

`lastErrorCode`
: Returns an integer representing the last OS error or nil if no error has occurred. Does not clear the error information

`lastErrorSymbol`
: Returns a Symbol representing the last OS error or nil if no error has occurred. Does not clears the error information.

GsSecureSocket

If one of the calls returns nil or false, you can determine the last error from an SSL function called from an instance method using the instance method:

`GsSecureSocket >> fetchLastIoErrorString`
: This fetches and clears the error string from a call to SSL functions for connect, accept, read, or write.

On the class side, the following methods return error strings for any SSL function call errors:

`GsSecureSocket class >> fetchErrorStringArray`
Returns an Array of error strings generated by the OpenSSL package. The errors returned are cleared from the SSL error queue. The array is ordered from oldest to newest error.

`GsSecureSocket class >> fetchLastCertificateVerificationError-ForClient`
`GsSecureSocket class >> fetchLastCertificateVerificationError-ForServer`
These methods fetches and clears a string representing the last certificate verification error logged by, respectively, a client SSL socket or a server SSL socket.

To clear the error queue, use the method

`GsSecureSocket class >> clearErrorQueue`

Chapter

12 Signals and Notifiers

This chapter discusses how to communicate between one session and another, and between one 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.

Other Signal-Related Issues
describes performance, signal buffer overflow, and other signal related considerations.

12.1 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 independent of 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.

12.2 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 one of 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.

The following section describes each of these steps in detail.

Setting Up a Notify Set

GemStone defines 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 an Object to a Notify Set

To add an object to your notify set, use 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, there are circumstances in which the both the root object and subobjects must appear in the notify set. For details, see "Special Classes" on page 225.

Example 12.1 creates a collection of stock holdings and then creates a notify set for the stocks in the collection. Finally, the session is set to automatically receive the notifier.

Example 12.1

```
"Create a Class to record stock name, number and price"
Object subclass: #Holding
   instVarNames: #('name' 'number' 'price')
   classVars: #()
   classInstVars: #()
   poolDictionaries: {}
   inDictionary: Published.

"Compile accessing methods"
Holding compileAccessingMethodsFor: Holding instVarNames.

"Add a Collection for Holdings to UserGlobals dictionary"
UserGlobals
   at: #MyHoldings put: IdentityBag new.

"Add some stocks to my collection"
MyHoldings add:
   (Holding new name: #USSteel; number: 1000; price: 50.00).
MyHoldings add:
   (Holding new name: #VMware; number: 50000; price: 95.00).
MyHoldings add:
   (Holding new name: #ATT; number: 100000; price: 30.00).

"Add the collection object to the notify set"
System addToNotifySet: MyHoldings.
(System notifySet) includesIdentical: MyHoldings.

"Enable receipt of signals"
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 special objects such as true, false, nil, and instances of Character, SmallInteger and SmallDouble.

Adding a Collection to a Notify Set

To add a collection of objects to your notify set, use 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; if you want to know when the elements change, you must add them to the notification set.

Example 12.2 shows the notify set containing both the collection object and the elements in the collection.

Example 12.2

```
"Add the stocks in the collection to the notify set"
System addAllToNotifySet: MyHoldings.
System notifySet.
%
an Array
  #1 a Holding
  #2 a Holding
  #3 a Holding

"Add the collection object itself to the notify set"
System addToNotifySet: MyHoldings.
System notifySet.
%
an Array
  #1 a Holding
  #2 a Holding
  #3 a Holding
  #4 an IdentityBag
```

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 thousands of objects 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 different system of change recording:

1. Have each session maintain its own list of the last several objects updated (a modify list). The list is a collection written only by that session.
2. Create a global collection of collections that contains every session's list of changes.
3. Put the global collection and its elements in your notify set, so you receive notification when a session commits a modified list of changed objects. Then you can check for changes of interest.

If the modify lists are ordered, this preserves the order of the additions, so that the new objects can be serviced in the correct order. Using the notifySet, 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 expression 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
```

Notification of New Objects

In a multi-user environment, objects are created in various sessions, committed, and immediately open to modification. It may not be sufficient to receive notifiers on the objects that existed at the beginning of your session. 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, to receive notification when the price of any stock in your portfolio changes, you can perform 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 all of these stocks in your notify set:

   ```
   System addAllToNotifySet: MyHoldings
   ```

3. Place the collection MyHoldings in your notify set, so that 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 the contents of MyHoldings have changed, compare the new MyHoldings with the original.
6. When you find new stocks, add them to your notify set, so that you will be notified if they are changed.

Example 12.3 shows one way to do steps 5 and 6.

Example 12.3

```
"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.

Note that only IdentityBag and its subclasses understand `"-"` as a difference operator.

Receiving Object Change Notification

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.

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 exception `ObjectsCommittedNotification`. By default, `ObjectsCommittedNotification` is disabled (except in GemBuilder for Smalltalk, which enables the signal as part of `GbsSession>>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 that the exception handler can take appropriate action.

The receiving session handles the notification with an exception handler. Your exception handler is responsible for reading the set of signaled objects (by sending the message `System class>>signaledObjects`) as well as taking the appropriate action.

```
ObjectsCommittedNotification addDefaultHandler:
   [:ex |
   | changes |
   changes := System signaledObjects.
   "do something with the changed objects"
   System enableSignaledObjectsError].
```

Reading the Set of Signaled Objects

The `System class>>signaledObjects` method reads the incoming changed object signals. This method returns an array, which 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.

Use a loop to call `signaledObjects` repeatedly, until it returns an empty collection. The empty collection guarantees that there are no more signals in the queue.

Also see the discussion of "Frequently Changing Objects" on page 224.

Polling for Changes to Objects

You also use `System class>>signaledObjects` to poll for changes to objects in your notify set.

Example 12.4 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 12.4

```
System disableSignaledObjectsError;
   signaledObjectsErrorStatus.
%

"Create a set."
Published at: #Changes put: IdentitySet new.
System commitTransaction.

System addToNotifySet: Changes.
%

"Login a separate session to perform the following"
Changes add: 'here is a change'.
System commitTransaction
%

"In the original session, see the signal"
| mySignaledObjs count |
System abortTransaction.
count := 0 .
[ mySignaledObjs := System signaledObjects.
mySignaledObjs size = 0 and:[ count < 50]
]
  whileTrue: [
   System sleep: 10 .
   count := count + 1
   ].
^ mySignaledObjs.
%
```

Troubleshooting

Notification on object changes may occasionally produce unexpected results. The following sections outline areas of concern.

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.

Common operations that trigger notification on the root object include:

- 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
- RcKeyValueDictionary

You can also have the problem with your own application classes. Wherever possible, you should implement objects so that changes modify the root object. You must also 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 12.5 shows a method that creates an object and automatically adds it to the notify set in the process.

Example 12.5

```
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
disableSignaledObjectsError
enableSignaledObjectsError
notifySet
removeAllFromNotifySet:
removeFromNotifySet:
signaledObjects
signaledObjectsErrorStatus
```

See Chapter 13, "Handling Exceptions", on page 233, for more on handling Exceptions such as ObjectsCommittedNotification.

12.3 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

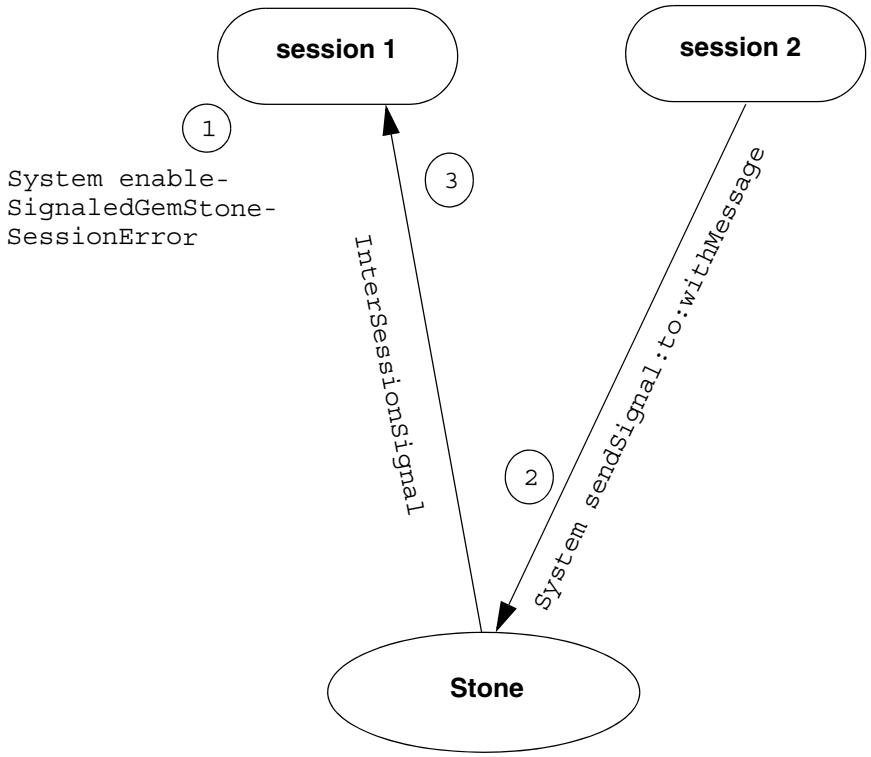
A signal is not an interrupt, and it does not automatically awaken an idle session. The signal can be received only when your session is actively executing Smalltalk code.

You can receive a signal from another session by polling for the signal or by receiving automatic notification.

As an example of Gem-to-Gem signaling, Figure 12.1 shows the following sequence of events:

1. session1 enables event signals from other Gem sessions. (For details, see "Receiving a Notification".)
2. session2 sends a signal to session1. (See “Receiving a Notification” on page 230.)
3. The Stone sends the exception `InterSessionSignal` to session1. The receiving session processes the signal with an exception handler. For details, see Chapter 13, Handling Exceptions.

Figure 12.1 Communicating from Session to Session



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 ID. To see a description of sessions that are currently logged in, execute the following method:

```
System currentSessions
```

This message returns an array of SmallIntegers representing session IDs for all current sessions. Example 12.6 shows how you might use this method to find the session ID for user1 and send a message.

Example 12.6

```
| sessionId serialNum 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'.
  signalToSend sendToSession: otherSession.
]
```

Example 12.6 uses the method `signalToSend sendToSession: otherSession`. Alternatively, you might use this method:

```
otherSession sendSignalObject: signalToSend
```

Still another alternative is this one, which replaces the final two expressions in Example 12.6 with a single expression:

```
System sendSignal: aSignalNumber to: otherSession withMessage:
aMessage
```

No matter how the message is sent, the other session needs to receive it, as shown in Example 12.7.

Example 12.7

```
GsSession currentSession signalFromSession message
%
reinvest form is here
```

Sending the Message

When you have the session ID, you can use the method `GsInterSessionSignal class>>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." An option is to create an application-level symbol dictionary of meanings for the different signal numbers.
- *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 12.8.

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 12.8, 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 12.8

```
System currentSessions do: [:each |
        System sendSignal: 8 to: each
               withMessage: 'new Holding: SallieMae'.].

System signalFromGemStoneSession at: 3.
```

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 notification 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 following sections.)

The method `System class>>signalFromGemStoneSession` 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 nil. This guarantees that there are no more signals in the queue. If signals are being sent quickly, you may not receive a separate `InterSessionSignal` 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 ID of the session that sent the signal (a SmallInteger).
- The signal value (a SmallInteger).

- The string containing the signal message.

If no signal has been sent, this method returns an empty array.

Example 12.9 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 12.9

```
| response count |
count := 0 .
[ response := System signalFromGemStoneSession.
  response size = 0 and:[ count < 50 ]
] whileTrue: [
   System sleep: 10.
   count := count + 1
   ].
^response
```

Receiving a Notification

To use the exception mechanism to receive signals from other Gem sessions, you must enable receipt of the `InterSessionSignal` notification. This exception has the same three arguments mentioned above:

- The session ID of the session that sent the signal (a SmallInteger).
- The signal value (a SmallInteger).
- The string containing the signal message.

By default, the `InterSessionSignal` notification is disabled, except in the GemBuilder for Smalltalk interface, which enables the error as part of `GbsSession>>gemSignalAction:`.

To enable this exception, execute:

```
System enableSignaledGemStoneSessionError
```

To disable the exception, send the message:

```
System disableSignaledGemStoneSessionError
```

To determine whether receiving this exception is presently enabled or disabled, send the message:

```
System signaledGemStoneSessionErrorStatus
```

This method returns true if the notification 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 that the exception handler can take appropriate action without further interruption. You must re-enable it afterwards.

12.4 Other Signal-Related Issues

GemStone notifiers and Gem-to-Gem signals use the same underlying implementation. The following performance and other considerations apply when using either mechanism.

Inactive Gem

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 be received only when GemBuilder is running, not when you are running client code, sitting at the Topaz prompt, waiting for activity on a socket, or waiting on a semaphore (as 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 GemStone Smalltalk application, you could set up a polling process that periodically sends out `GbsSession>>pollForSignal`. The `pollForSignal` method causes GemBuilder for Smalltalk to poll the repository. GemBuilder for C also provides a 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.

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 222.

Each session has a signal buffer that will accommodate 50 signals. Signals remain in the signal buffer until they are received and read by the receiving session. If the receiving session does not read the signals, or if it does not read them fast enough to keep up with signals that are being sent, the signal buffer will fill up. In this case, further signals will cause the Exception `SignalBufferFull` to be signalled on the sender. Set your application so that 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 nil. 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. For a better idea of how the message queue works, see `System class>>sendSignal:to:withMessage:` in the image.

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 11, "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. For example, 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 12.10 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 12.10

```
| 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).
(GsFile openWriteOnServer: '$GEMSTONE/gemmessage.txt')
      addAll: logString; close.
```

Chapter

13 Handling Exceptions

GemStone Smalltalk implements the ANSI exception handling protocols, with provisions for signaling that an exception has occurred and for defining handlers for signaled exceptions.

The Exception Class Hierarchy
describes the exception class hierarchy, listing the subclasses that correspond to events that you may want to handle.

Signaling Exceptions
describes the mechanism whereby an application can signal that a some notable event occurred. The class of the signaled exception determines which handler(s) will be invoked. A handler might halt execution and report an error to the user.

Handling Exceptions
describes how to define handlers in your application to cope with signaled exceptions. Depending on the type of the exception, your application might be able to handle the exception gracefully, possibly even without the user being informed of the exception.

The Legacy Exception Handling Framework
describes the legacy exception handling framework.

13.1 The Exception Class Hierarchy

GemStone/S 64 Bit supports the ANSI Exception framework. The ANSI Exception framework defines subclasses to match the granularity of errors that you may want to handle.

GemStone also supports a Legacy Exception framework, for compatibility with earlier version of Gemstone. This can be used to signal and handle ANSI exceptions. The Legacy Exception framework is described starting on page page 242.

Figure 13.1 shows the ANSI exception handler class hierarchy.

Figure 13.1 Exception Class Hierarchy

```
AbstractException ( gsResumable gsTrappable gsNumber currGsHandler
                gsStack gsReason gsDetails tag messageText gsArgs )
   Exception
        ControlInterrupt
           Break
           Breakpoint  ( context stepPoint )
           ClientForwarderSend  ( receiver clientObj selector )
           Halt
           TerminateProcess
        Error
           CompileError
           EndOfStream
           ExternalError
              IOError
                 SocketError
              SystemCallError  ( errno )
           GciError
           GciLegacyError
           GsMalformedQueryExpressionError
           ImproperOperation  ( object )
              ArgumentError
              ArgumentTypeError  ( expectedClass actualArg )
              CannotReturn
              LookupError  ( key )
              OffsetError  ( maximum actual )
              OutOfRange  ( minimum maximum actual )
                 FloatingPointError
              RegexpError
           IndexingErrorPreventingCommit
           InternalError
              GciTransportError
           LockError  ( object )
           NameError  ( selector )
              MessageNotUnderstood  ( envId receiver )
           NumericError
              ZeroDivide  ( dividend )
           RepositoryError
           SecurityError
           SignalBufferFull
           ThreadError
           TransactionError
           UncontinuableError
           UserDefinedError
        Notification
           Admonition
              AlmostOutOfMemory
              AlmostOutOfStack
              RepositoryViewLost
           Deprecated
           FloatingPointException
           GsUnsatisfiableQueryNotification
           InterSessionSignal  ( sendingSession signal )
           ObjectsCommittedNotification
           TransactionBacklog  ( inTransaction )
           Warning
              CompileWarning
        TestFailure
           ResumableTestFailure
```

13.2 Signaling Exceptions

ANSI Exceptions are *class-based*: you use a class in the Exception hierarchy to describe errors and other exceptions in your GemStone Smalltalk programs.

You can extend the built-in exception types by defining new subclasses. You can also change your new exception's default behavior by adding method overrides to the new class (for example, `defaultAction` and `isResumable`).

The ANSI exception handling framework provides for zero or more dynamic (stack-based) handlers and a list of zero or more default handlers, ordered in the sequence they were installed.

When an application sends a message of the form:

```
Exception signal: aString
```

GemStone Smalltalk creates an *instance* of the signaled class and performs the following search for a suitable handler:

1. Search the stack for a handler associated with the exception class. In a dynamic (stack-based) handler (page 236), you explicitly identify a block of application code that might signal an exception to which you wish to respond.
2. Search the default (static) handlers. A default handler (page 240) is invoked if a dynamic handler is not found or if the last dynamic handler passes the exception.
3. Search the exception class for an implementation of the instance method `defaultAction`. Some exception classes redefine this method, thereby establishing a handler to use in the case that there is no suitable dynamic or default handler or if the last such handler passes the exception. For example, with Notification, the default action is to ignore the exception.

   If the exception class does not override the implementation of `defaultAction` in class AbstractException, halt the GemStone Smalltalk interpreter and pass the exception back to the client to be handled (by Topaz, GemBuilder, or another application) as an error.

Example 13.1

```
method: Employee
age: anInt
(anInt between: 15 and: 65)
   ifFalse: [Error signal: 'Employee age out of range'].
age := anInt.
%
```

13.3 Handling Exceptions

Other than a few fatal errors, most signaled exceptions can be handled in your GemStone Smalltalk application. To do so, you identify the type of exception that might be signaled (Exception or, more often, a subclass of Exception) and provide GemStone Smalltalk code to handle the exception.

GemStone Smalltalk allows you to define two kinds of exception handlers: *dynamic (stack-based) handlers* and *default (static) handlers.*

Dynamic (Stack-Based) Handlers

A dynamic (stack-based) handler is associated with an executable block (instance of ExecBlock) and the associated state in which the GemStone Smalltalk virtual machine is presently executing. These handlers live and die with their associated blocks—when the block is exited, the handler is gone.

A dynamic handler is associated with exactly one ExecBlock and applies as long as the ExecBlock is being executed. Because an ExecBlock can be embedded in another ExecBlock (either directly or via another method), multiple dynamic handlers can be active at one time. Figure 13.2 illustrates this relationship.

Figure 13.2 ExecBlock and Associated Handlers

```
           top
           ...
           ^
    /      ExecBlock -> Optional dynamic handler
Stack      ^
    \      ExecBlock -> Optional dynamic handler
           ^
           bottom
```

To define a dynamic handler for an ExecBlock, send the `on:do:` message to the block. Example 13.2 defines an `averagePay` method for the Employee class. The method calculates an average by dividing two values. If the division signals a ZeroDivide exception, the exception handler returns zero as the result of the method. In this implementation, the method will never result in a "division by zero error" being seen by the user. (Of course, there are other ways you might write this particular method. This example simply serves to highlight the `on:do:` exception handling approach.)

Example 13.2

```
method: Employee
averagePay

   [
      ^self totalPay / self yearsOfService.
   ] on: ZeroDivide do: [:ex |
      ^0.
   ].
%
```

The first argument to the `on:do:` method specifies what types of exception the handler should catch. The argument can be a class in the Exception hierarchy, or it can be an ExceptionSet made up of one or more classes in the Exception hierarchy.

The second argument specifies a one-argument ExecBlock that will be invoked when the specified exception is signaled. The one argument is the newly-created instance of the class of the exception that was signaled, and can contain additional information about the exception (including the string that was passed to the `signal:` method). For example, an instance of the ZeroDivide error can be queried for the dividend (obviously, the divisor is zero). Similarly, an instance of the MessageNotUnderstood error can be queried for the receiver and message (selector and arguments).

Selecting a Handler

When an exception is signaled, GemStone starts at the top of the current process's stack, searching down the stack for a handler that handles the exception. Each exception handler in the stack is examined to see if it was installed (using the `on:do:` message) as a handler for the signaled exception's class. If a handler is found but it does not handle the signaled exception, it is passed over and the search continues down the stack.

A handler for a superclass will handle subclass exceptions. That is, an exception handler for the class Error will be invoked for an exception of its subclass ZeroDivide, and an exception handler for the class Notification will be invoked for an exception of its subclass Warning.

A subclass does not, however, handle a superclass exception. This means that an exception handler for the class MessageNotUnderstood will not be invoked for an exception of its superclass Error.

Example 13.3 contains six blocks, three protected blocks and three handler blocks. Each of the three `on:do:` messages creates a new stack frame that has an associated handler block.

Example 13.3

```
method: Employee
doStuff

   | a b c |
   a := [
      self doStuffA.
      b := [
         self doStuffB.
         c := [
         self doStuffC.
         self doStuffD.
         ] on: ZeroDivide do: [:zdEx |
         self handleZeroDivide: zdEx.
         ^self.
         ].
         self doStuffE.
      ] on: Warning do: [:wEx |
         self handleWarning: wEx.
         wEx resume: #ok.
      ].
      self doStuffF.
      #good.
   ] on: Error do: [:erEx |
      self handleError: erEx.
      erEx return: #bad.
   ].
%
```

As shown in Figure 13.3, the handler for Error is installed first, and catches any Error or subclass exception signaled during the block that begins with `self doStuffA`.

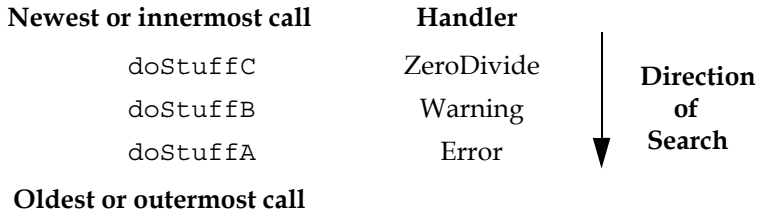
The handler for Warning is installed next, and catches any Warning or subclass exception signaled during the block that begins with `self doStuffB`.

If a ZeroDivide error is signaled during `doStuffB`, it is handled by the Error handler, not by the ZeroDivide handler (which is not yet installed).

The handler for ZeroDivide is installed last, and catches any ZeroDivide error or subclass exception signaled during the block that begins with `self doStuffC`.

If a MessageNotUnderstood error were signaled during `doStuffC`, it would not be handled by either the ZeroDivide or Warning handler, even though they were installed more recently. Those handlers are not of the proper class; MessageNotUnderstood does not inherit from ZeroDivide or Warning. Instead, a MessageNotUnderstood error would be handled by the Error handler associated with the block that begins with `self doStuffA`.

Figure 13.3 Selecting a Handler



Flow of Control

Once control is passed by sending `value:` to the handler block with the exception instance as an argument, the handler block can attempt to address the situation.

Keep in mind that a dynamic handler is just an ExecBlock that is defined in a method and passed as an argument during a message send (like a block sent with a `select:` message). As such, the dynamic handler has access to the method context in which it is defined, including method temporaries and block variables in its scope, as well as the object in which the method is defined (including instance variables). The handler may, of course, send messages to any object to which it has access.

In particular, the dynamic handler may return from the method containing the dynamic handler. In Example 13.3 (on page 238), the ZeroDivide handler returns `self`. If a ZeroDivide exception were signaled during `doStuffC`, then the `doStuff` method would return and other messages would never be sent (`doStuffD`, `doStuffE`, and `doStuffF`).

Messages That Alter the Flow of Control

In addition to an explicit return from the containing method, a dynamic handler can send the following messages to the exception instance to cause other changes in the flow of control. Sending one of these messages is similar to a method return in that there is no return from these messages (except for `outer`, which might return).

`resume:` *anObject*
: Causes *anObject* to be returned as the result of the `signal:` message that triggered the exception. Sending `resume:` to a non-resumable exception is an error.

  In Example 13.3, the Warning handler returns `#ok` as the result of the `signal:` message.

`resume`
: Causes nil to be returned as the result of the `signal:` message. Sending `resume` to a non-resumable exception is an error.

`return:` *anObject*
: Causes *anObject* to be returned as the result of the `on:do:` message to the protected block. In Example 13.3, the Error handler returns `#bad` to the local variable 'a' as the result of the `on:do:` message. If no Error occurred during the protected block, then the `on:do:` method would return `#good` as the result of evaluating the protected block.

`return`
Causes nil to be returned as a result of the `on:do:` message.

`retry`
Unwinds the stack and re-evaluates the protected block (by sending the `on:do:` message again).

`retryUsing:` *aBlock*
Unwinds the stack and evaluates the replacement block as the protected block, sending it the `on:do:` message.

`pass`
Exits the current handler and searches for the next handler. In Example 13.3, if the ZeroDivide handler sends `pass` to the ZeroDivide exception instance, control passes to the Error handler as if the ZeroDivide handler didn't exist (except that any side effects of its operation up to the `pass` message are preserved).

`outer`
Similar to `pass`, except that if the outer handler sends `resume:` or `resume` to the exception instance, control returns to the inner handler from the `outer` message.

`resignalAs:` *replacementException*
Sending this message causes GemStone Smalltalk to start searching for an exception handler for *replacementException* at the top of the stack as if the original `signal:` message had been sent to *replacementException* instead of the receiver.

NOTE

If none of the above messages are sent to alter the flow of control, the value of the last expression in the block will be returned as the result of the `on:do:` *message. (For clarity, you could make this behavior explicit by using the* `return:` *message.)*

Default Handlers

As described above, a dynamic (stack-based) handler protects a particular block of code that exists in the same method as the handler. This is appropriate when you only want to handle a particular exception during execution of the protected code. When the protected block finishes executing, the handler is no longer in effect.

There are, however, other exceptions that could happen at any time for reasons entirely unrelated to your code — for example, being notified that the disk is full (RepositoryError) or that another Gem is sending you a signal (InterSessionSignal). For such exceptions, you can establish a default (or static) handler.

Since ANSI does not provide a direct API for adding and removing default handlers at runtime, GemStone provides the following methods to deal with default handlers in the context of the ANSI framework.

`Exception class >> addDefaultHandler:` *aOneArgumentBlock*
Returns a GsExceptionHandler that understands the message `remove` and adds the new handler to the beginning of the `defaultHandlers` list. After *aOneArgumentBlock* (equivalent to the second argument to `on:do:`) is invoked, the argument (an instance of Exception or one of its subclasses) responds appropriately to `pass` and `outer` seamlessly between stack-based and default handlers.

`AbstractException class >> defaultHandlers`
Returns a SequenceableCollection (or subclass) of GsExceptionHandler instances that will catch instances of the receiver (typically, a subclass of AbstractException). The result does not include any legacy static handlers (as discussed on page 243). This collection may be empty and typically is a subset of the installed default (static) handlers.

`GsExceptionHandler >> remove`
Since a default handler is not tied to a specific block of code, once installed it remains in effect until explicitly removed (or until the session logs out). This method removes (and returns) the default handler if it is found. If it is not found, returns nil.

Default Actions

The third line of defense for an exception (after dynamic and default handlers) occurs when the virtual machine sends the message `defaultAction` to the signaled exception. Because `defaultAction` is implemented in AbstractException, every exception will eventually be handled. The ultimate default action (in AbstractException) is to stop the GemStone Smalltalk interpreter and pass the exception back to the client (to be handled by Topaz, GemBuilder, or another application).

Exception subclasses can override this method to provide alternate behavior. For example, the default action for Notification is to ignore the notification and return nil from the `signal:` message. For Deprecated, the default action is to log information; for MessageNotUnderstood, the default action is to retry the original action.

To define a default handler for a new exception, add a `defaultAction` method to your new exception class.

13.4 The Legacy Exception Handling Framework

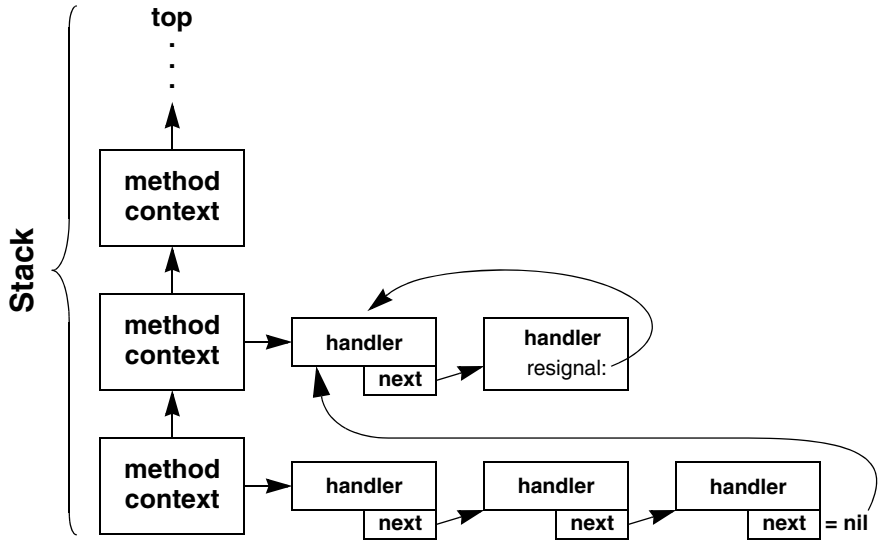
ANSI exception handling, as described previously, is the primary mechanism for dealing with errors in your programs. The legacy handler protocol is deprecated, and all exceptions are now raised as ANSI exceptions. While we strongly encourage the use of ANSI protocol, legacy protocol may be used to raise and handle ANSI exceptions.

Dynamic (Stack-Based) Exception Handler

In ANSI, a dynamic (stack-based) exception handler is associated with an ExecBlock. By contrast, a dynamic legacy exception handler is associated with a method being executed. These exception handlers live and die with their associated method contexts—when the method returns, control is passed to the next method and the exception handler is gone.

Each exception handler is associated with one method context, but each method context can have a stack of associated exception handlers. The relationship is diagrammed in Figure 13.4.

Figure 13.4 Method Contexts and Associated Exceptions



Installing a Dynamic (Stack-Based) Exception Handler

To define a legacy dynamic (stack-based) handler for an exception, use the class method `Exception category:number:do:`.

- The argument to the `category:` keyword is ignored.
- The argument to the `number:` keyword is the specific error number you wish to catch, which can be nil (to catch all exceptions).

- 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 that was signaled.
  - The second argument to the four-argument block is always GemStoneError.
  - The third argument to the four-argument block is an error number.
  - The fourth argument to the four-argument block is the data passed in when invoking the error.

If your exception handler does not specify an error number (an error number of nil), then it receives control in the event of any exception.

The exception handler in Example 13.4 catches the GemStone exception ZeroDivide and returns either PlusInfinity or MinusInfinity, depending on the sign of the dividend.

Example 13.4

```
| a b c |
a := 0.
Exception
      category: GemStoneError
      number: 2026
      do: [:ex :cat :num :args |
         "Return a value as a result of the #'/' message"
         ex dividend * 1.0e0 / 0].

"This might give a ZeroDivide error,
depending on the value of a"
b := -10 / a.
c := b * 3.
c
```

NOTE

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.

Default (Static) Exception Handlers

A *default (static) exception handler* 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 handler 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, such as the errors listed in Table 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.

Installing a Default (Static) Exception Handler

To define a default (static) exception handler, use the Exception class method `installStaticException:category:number:`.

- 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 ignored.
- The argument to the `number:` keyword is the specific error number you wish to catch.

The following exception handler, for example, handles the error #abortErrLostOtRoot:

Example 13.5

```
UserGlobals at: #tx3 put:
   ( "Handle lost OT root"
     Exception
       installStaticException: [:ex :cat :num :args |
         System abortTransaction.
       ]
      category: nil
      number: 3031
     subtype: nil
   ).
```

To remove the handler, execute:

```
self removeExceptionHandler: (UserGlobals at: #tx3).
```

GemStone Event Exceptions

The errors in Table 1 are sometimes called *event exceptions*. Although they are not true errors, their implementation is based on the GemStone error mechanism. For examples that use these event exceptions, also called signals, see Chapter 12, "Signals and Notifiers".

In Table 1, the legacy error symbol (and number) is listed along with the corresponding GemStone/S 64 Bit v3.2 exception class.

NOTE

The array LegacyErrNumMap (in Globals) describes the mapping of legacy (pre-3.0) error numbers to ANSI exception classes (as described in Chapter 13).

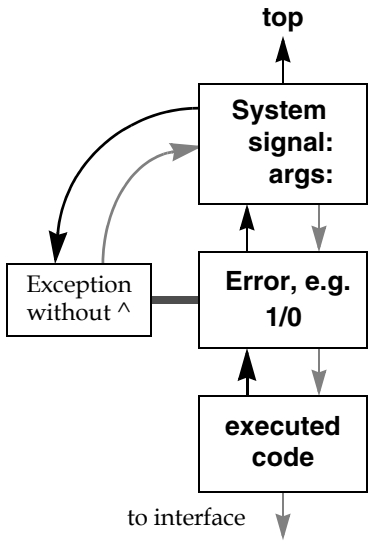
Table 1 Common GemStone Event Exceptions

Exception class Legacy symbol (and number)	Description
TransactionBacklog #*rtErrSignalAbort (6009)* #*rtErrSignalFinishTransaction (6012)*	When `System inTransaction` returns false (running outside a transaction), Stone requested Gem to abort. This error is generated only if you have executed either **`System en`**`ableSignaledAbortError` or **`TransactionBacklog enableSignalling`**. When `System inTransaction` returns true (the session is in transaction), Stone has requested the session to commit, abort, or continue (with continueTransaction) the current transaction. This error is received only if you have executed either `System enableSignaledFinishTransactionError` or **`TransactionBacklog enableSignalling`**.
ObjectsCommittedNotification #*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 either `System enableSignaledObjectsError` or **`ObjectsCommittedNotification enableSignalling`**
InterSessionSignal #*rtErrSignalGemStoneSession (6010)*	Your session received a signal from another GemStone session. This error is received only if you have executed either `System enableSignaledGemstoneSessionError` or **`InterSessionSignal enableSignalling`**. InterSessionSignal arguments: 1. The session ID of the session that sent the signal. 2. An integer representing the signal. 3. A message string.
AlmostOutOfMemory #*rtErrSignalAlmostOutOfMemory (6013)*	Temporary object memory for the session is almost full. The error is deferred if in user action or index maintenance. This error is enabled by default, but the default handler has no action. After a signal is received, it must be reenabled using `System enableAlmostOutOfMemoryError` or **`AlmostOutOfMemory enable`**.
RepositoryError #*rtErrTranlogDirFull (2339)*	All available transaction log directories or partitions are full. This error is received if you are DataCurator or SystemUser, otherwise only if you have executed `System enableSignalTranlogsFull`.
RepositoryViewLost #*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.

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. You must write all default (static) exception handlers this way, because the stack usually changes by the time they catch an error. Dynamic (stack-based) exception handlers can also be written to behave that way, like the one in Example 13.4 on page 243. See Figure 13.5.

Figure 13.5 Default Flow of Control in Legacy Exception Handlers



Sometimes, however, this is not useful behavior—the application may simply have to raise the same error again. In dynamic (stack-based) exception handlers, it can be useful instead to return control to the method that defined the handler.

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 method in Example 13.1 redefines how the GemStone exception #ZeroDivide is to be handled.

Example 13.1

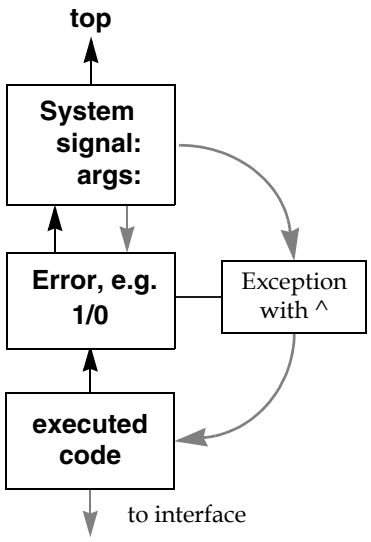
```
| a b c |
a := 0.
Exception
   category: GemStoneError
   number: 2026
   do: [:ex :cat :num :args |
   "Return from this method with a String"
   ^'zero divide'
   ].

"When a is zero, the error will be caught and the method will
return without assigning any value to b or c"
```

```
b := -10 / a.
c := b * 3.
c
```

Figure 13.6 shows the flow of control in Example 13.1.

Figure 13.6 Dynamic (Stack-Based) Exception Handler with Explicit Return



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 ignored.
- 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 might be used to build the error message.

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 in the previous section, `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 execute the following code:

Example 13.2

```
| x y |
Exception
        category: GemStoneError
        number: 2026
        do: [:ex :cat :num :args | ex remove. 'result of first handler'].
Exception
        category: GemStoneError
        number: 2026
        do: [:ex :cat :num :args | ex remove. 'result of second handler'].
x := 1 / 0. "handled by the second (most recent) handler"
y := 2 / 0. "handled by the first handler; the second was removed"
Array with: x with: y.

" anArray( 'result of second handler', 'result of first handler')"
```

The first occurrence of the error executes the most recent exception defined. 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 an array of two strings, as shown here.

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.

Raising Exceptions

Legacy methods for raising exceptions can be used, but raise ANSI 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 are passed to the handler.
- The argument to the `signalDictionary:` keyword is ignored.

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. Alternatively, 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, as described above.

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.

ANSI Integration

The ANSI and legacy frameworks should work together so that signaling an ANSI exception is caught by a legacy exception handler. Example 13.3 shows a sample use of a legacy handler to catch signaled ANSI exceptions.

Example 13.3

```
method: Employee
legacyMethod

   self doA.
   "Install a legacy handler"
   Exception
      category: nil
      number: nil
      do: [:ex :cat :num :args |
         self handlerCode.
         self shouldReturn ifTrue: [
            ^self returnValue.
         ].
         self continueValue.
      ].
   self doB.
   "Signal an ANSI error"
   instVar1 := Error signal: 'something bad happened!'.
   self doC.
   ^instVar2.
%
```

When this method is invoked, it calls `doA` before installing the exception handler. After the exception handler is installed, the method calls `doB`. If any exception is signaled during the execution of `doB`, the handler is invoked.

Next, an explicit error is invoked, using the ANSI protocol. This signaled ANSI exception is caught by the legacy exception handler installed earlier in the method. After evaluating the `handlerCode`, the handler decides whether to return from the method or continue. If it returns, the result of `returnValue` is returned. If it continues, the result of `continueValue` is stored in `instVar1`, and the method proceeds with `doC` and finally returns `instVar2`.

Chapter

14 Performance and Optimization

GemStone Smalltalk includes several tools to help you tune your applications for faster performance.

Clustering Objects for Faster Retrieval
How to cluster objects that are often accessed together so that many of them can be found in the same disk access.

Profiling Smalltalk Execution
Profiling tools that allow you to pinpoint the problem areas in your application code.

Modifying Cache Sizes for Better Performance
How to increase or decrease the size of various caches in order to minimize disk access and storage reclamation.

Managing VM Memory
Issues to consider when managing temporary object memory, and presents techniques for diagnosing and addressing OutOfMemory conditions.

NotTranloggedGlobals
Optimize certain operations by avoiding writing tranlog entries.

Other Optimization Hints
Allow operations on large collections without using temporary object memory.

14.1 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.

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. You can use GemStone's VSD utility to find out how many times your application is accessing the disk. VSD allows you to chart system statistics over time to better understand the performance of your system. See the *VSD User's Guide* for more information on using VSD.

The following statistics are of interest:

- `pageReads` — how many pages your session has read from the disk since the session began
- `pageWrites` — how many pages your session has written to the disk since the session began

You can examine the values of these statistics before and after you commit each transaction to discover how many pages it read in order to perform a particular query, and to determine the number of disk accesses required by the process of committing the transaction.

It is tempting to ignore these issues until you experience a problem such as an extremely slow application, but if 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 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.

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:

- `AllClusterBuckets at: 4` is the bucket used to cluster the methods associated with kernel classes.
- `AllClusterBuckets at: 5` is the bucket used to cluster the strings that define source code for kernel classes.
- `AllClusterBuckets at: 6` 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 bucket's 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 the predefined instances of ClusterBucket. 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. 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. Alternatively, 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
```

(Recall that the first seven elements of the array are predefined.)

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 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. Cluster buckets do not contain or reference the objects clustered on them -- the objects that are clustered keep track of their bucket. This also avoids problems with authorizations.

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.

To avoid concurrency conflicts, you should design your clustering when you design your application. 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 14.1

```
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 or clusterId, writes that cluster bucket and thus 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 instance of subclasses of UnorderedCollection are created and modified using the cluster bucket associated with the specific collection, if any. To change the clustering of an indexed 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. Another method 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 as shown in Example 14.2.

Example 14.2

```
Object subclass: 'Name'
   instVarNames: #('first' 'middle' 'last')
   classVars: #( )
   classInstVars: #()
```

```
   poolDictionaries: {}
   inDictionary: UserGlobals.
Object subclass: 'Employee'
   instVarNames: #('name' 'job' 'age' 'address')
   classVars: #( )
   classInstVars: #()
   poolDictionaries: {}
   inDictionary: UserGlobals.
```

The following clustering method might be suitable 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 14.3

```
method: Employee
clusterBreadthFirst
   self cluster.
   name cluster.
   job cluster.
   address cluster.
   name first cluster.
   name middle cluster.
   name last cluster.
   ^false
%

| Lurleen |
Lurleen := Employee new name: (Name new first: #Lurleen);
   job: 'busdriver'; age: 24; address: '540 E. Sixth'.
Lurleen clusterBreadthFirst
%
```

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 special objects (instances of SmallInteger, Character, Boolean, SmallDouble, or UndefinedObject) has no effect. Hence no clustering message is sent to *age* in the previous example.

After sending `cluster` to an Employee, the Employee is clustered as follows:

```
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 a special 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.

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 is 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

Both `cluster` and `clusterDepthFirst` 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 expression 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 and Memory Use

Clustering tags objects in memory so that when the next successful commit occurs, the objects are clustered onto data pages according to the method specified. After an object has been clustered, it is considered to be "dirty". If you cluster a large number of objects, you may need to increase temporary object memory to avoid running out of session memory. See "Managing VM Memory" on page 270.

Using Several Cluster Buckets

When you want to write a loop that clusters parts of each object in a group into separate pages, it is helpful to have multiple cluster buckets available. Suppose that you had defined class SetOfEmployees and class Employee as in Chapter 4. Suppose, in addition, that you wanted a clustering method to write all employees contiguously and then write

all employee addresses contiguously. With only one cluster bucket at your disposal, you would need to define your clustering method as shown in Example 14.4. In this approach, each employee is fetched once for clustering, then fetched again in order to cluster the employee's address.

Example 14.4

```
method: SetOfEmployees
clusterEmployees
  self do: [:n | n cluster].
  self do: [:n | n address cluster].
%
myEmployees clusterEmployees
```

Clustering Class Objects

Clustering provides the most benefit for small groups of objects that are often accessed together — for example, a class with its instance variables. Those instance variables of a class that describe the class's variables 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 1 Clustering Protocol

`clusterBehavior`	Clusters in depth-first order the parts of the receiver required for executing GemStone Smalltalk code (the receiver and its method dictionary).
`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*, and *class histories*.
`clusterBehaviorExcept-Methods:`*aCollectionOfMethod-Names*	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 frequently used methods to be packed more densely.

The code in Example 14.1 clusters class Employee's structure-describing variables, then its class methods, and finally its instance methods.

Example 14.1

```
| behaviorBucket descriptionBucket |
behaviorBucket := AllClusterBuckets at: 4.
descriptionBucket := AllClusterBuckets at: 5.
System clusterBucket: descriptionBucket.
Employee clusterDescription.
System clusterBucket: behaviorBucket.
Employee class clusterBehavior.
Employee clusterBehavior.
%
```

The following 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 clustered in the same way forever. If you edit some of the objects in the data structure, the edited object will be placed on a new page in the same clusterBucket. The performance benefit of clustering is that the objects are on the same page, but since the clusterBucket will span multiple pages, the objects may be in the same clusterBucket but not on the same page.

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, as discussed in the next section.

For special objects (instances of SmallInteger, Character, Boolean, SmallDouble, or UndefinedObject), the page number returned is 0.

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 space.

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.

14.2 Profiling Smalltalk Execution

Ordinarily, disk access has the greatest impact on application performance. However, your GemStone Smalltalk code can also affect the speed of your application; as with other programming languages, some code is more efficient than other code. To help you determine how you can best optimize your application, GemStone Smalltalk provides a profiling tool, defined by the classes ProfMonitor and its subclass ProfMonitorTree.

System >> millisecondsToRun:

Keep in mind that if you simply want to know how long it takes a given block to return its value, you can use the familiar GemStone Smalltalk method `System millisecondsToRun:` *aBlock*. This method takes a zero-argument block as its argument and returns the time in milliseconds required to evaluate the block.

Classes ProfMonitor and ProfMonitorTree

ProfMonitor and ProfMonitorTree are classes that allow 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. When an instance of one of these classes starts profiling, it will take a method call stack 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.

The reports returned by ProfMonitorTree include a execution tree structure as well as the reports returned by ProfMonitor; otherwise these classes have the same interface. This discussion use ProfMonitor to refer to either class.

ProfMonitor, by default, will take a sample every millisecond (1 ms). You can specify the interval at which ProfMonitor takes samples using the instance methods `interval:` or `intervalNs:`, or class method with these keywords. Options with `Ns:` specify the interval in nanoseconds; a nanosecond is a billionth of a second. The minimum interval is 1000 nanoseconds.

It may be convenient to refer to Table 1 when determining the sample interval and reading the results:

Table 1 Subsecond time conversions

seconds	milliseconds ms	microseconds us	nanoseconds ns
1	1000	1,000,000	1,000,000,000
	1	1000	1,000,000
		1	1000

By default, ProfMonitor reports every method it found executing. It may be more useful to limit the reporting to methods that execute at least some number of times, to reduce clutter in the results. To do this, set the lower limit using the instance method `reportDownTo:` *anInteger* or methods with the keyword `downTo:`.

ProfMonitor stores its results temporarily in a file with the default filename /tmp/gemprofile.tmp. You can specify a different filename by using ProfMonitor's

instance method `fileName:`. This file is deleted by profiling block methods, profileOff, and reportAfterRun* methods.

Profiling Your Code

Profiling a Block of Code

ProfMonitor provides several methods that allow you to profile a block of code and report the results with a single class method.

By default, profiling uses a sampling interval of 1 ms, and includes every method it finds in its results, even those executing only once.

```
ProfMonitorTree
   monitorBlock: [ 100 timesRepeat:
      [ System myUserProfile dictionaryNames ] ]
```

This example uses a 5 ms sampling interval to 5 ms, and includes only methods that were found executing more than once:

```
ProfMonitorTree
   monitorBlock: [ 100 timesRepeat:
      [ System myUserProfile dictionaryNames ]]
   downTo: 2
   interval: 5
```

This example samples every 5000 ns (5 us), so a much more detailed profile report will be produced for the same code block; it reports only methods that were sampled 10 times or more.

```
ProfMonitorTree
   monitorBlock: [ 100 timesRepeat:
      [ System myUserProfile dictionaryNames ]]
   downTo: 10
   intervalNs: 5000
```

Multi-Step Profiling

You can also explicitly start and stop profiling, allowing you to profile any arbitrary sequence of GemStone Smalltalk statements, rather than only blocks of code.

To start and stop profiling, you can use the class method `profileOn`, which create an instances of ProfMonitor and starts profiling; when you are done, the instance method `profileOff` stops profiling and reports the results.

For example:

```
run
UserGlobals at: #myMonitor put: ProfMonitorTree profileOn.
%

run
100 timesRepeat: [ System myUserProfile dictionaryNames ].
%

run
(UserGlobals at: #myMonitor) profileOff.
%
```

Profiling beyond default variables

You can also create and configure the instance of ProfMonitor. This allows you full control, including enabling features such as object creation profiling that are not accessible from the default interface.

To profile in this way you will perform the following steps:

Step 1. Create instance using `ProfMonitor new`.

Step 2. Configure it as desired, using instance methods `interval:`, `intervalNs:`, and/or `traceObjectCreation:`.

Step 3. start profiling using the instance method `startMonitoring`.

Step 4. execute your code.

Step 5. stop profiling using the instance method `stopMonitoring`.

Step 6. gather results and report, using `reportAfterRun` or `reportAfterRunDownTo:`.

For example:

```
| aMonitor |
aMonitor := ProfMonitorTree new.
aMonitor interval: 2.
aMonitor traceObjectCreation: true.
aMonitor startMonitoring.
100 timesRepeat: [ System myUserProfile dictionaryNames ].
aMonitor stopMonitoring.
aMonitor reportAfterRun.
```

The Profile Report

The profiling methods discussed in previously return a string formatted as a report. The following example shows a sample run and the resulting report.

Example 14.1

```
topaz 1> printit
ProfMonitorTree
        monitorBlock:[
                200 timesRepeat:[ System myUserProfile dictionaryNames] ]
        downTo: 2
%
================
STATISTICAL SAMPLING RESULTS
elapsed CPU time:    90 ms
monitoring interval: 1.0 ms

 tally       %   class and method name
------    -----   -------------------------------------
    23   24.21   Array                       >> _at:
    22   23.16   IdentityDictionary          >> associationsDo:
    18   18.95   block in SymbolList         >> names
    18   18.95   AbstractDictionary          >> _at:
    11   11.58   block in AbstractDictionary >> associationsDetect:ifNone:
     2    2.11   Object                      >> _basicSize
     1    1.05   11 other methods
    95  100.00   Total


================
STATISTICAL STACK SAMPLING RESULTS
elapsed CPU time:    90 ms
monitoring interval: 1.0 ms

 total       %   class and method name
------    -----   -------------------------------------
    95  100.00   GsNMethod class             >> _gsReturnToC
    95  100.00   executed code
    95  100.00   ProfMonitor class           >> monitorBlock:downTo:
    95  100.00   ProfMonitor                 >> monitorBlock:
    94   98.95   block in executed code
    94   98.95   UserProfile                 >> dictionaryNames
    94   98.95   SymbolList                  >> namesReport
    94   98.95   SymbolList                  >> names
    94   98.95   AbstractDictionary          >> associationsDetect:ifNone:
    94   98.95   IdentityDictionary          >> associationsDo:
    29   30.53   block in AbstractDictionary >> associationsDetect:ifNone:
    23   24.21   Array                       >> _at:
    18   18.95   block in SymbolList         >> names
    18   18.95   AbstractDictionary          >> _at:
     2    2.11   Object                      >> _basicSize
     1    1.05   2 other methods
    95  100.00   Total
```

```
================
STATISTICAL METHOD SENDERS RESULTS
elapsed CPU time:    90 ms
monitoring interval: 1.0 ms

     %       %                     Parent
  self  total   total  local  Method
  Time   Time      ms    %         Child
------ ------  ------  -----  -----------

=    0.0  100.0    90.0    0.0  GsNMethod class >> _gsReturnToC
                   90.0  100.0       executed code
-------------------------------------------------------

                   90.0  100.0       GsNMethod class >> _gsReturnToC
=    0.0  100.0    90.0    0.0  executed code
                   90.0  100.0       ProfMonitor class >> monitorBlock:downTo:
-------------------------------------------------------

                   90.0  100.0       executed code
=    0.0  100.0    90.0    0.0  ProfMonitor class >> monitorBlock:downTo:
                   90.0  100.0       ProfMonitor  >> monitorBlock:
-------------------------------------------------------

                   90.0  100.0       ProfMonitor class >> monitorBlock:downTo:
=    0.0  100.0    90.0    0.0  ProfMonitor  >> monitorBlock:
                   89.1   98.9       block in executed code
                    0.9    1.1       ProfMonitor  >> startMonitoring
-------------------------------------------------------

                   89.1  100.0       ProfMonitor  >> monitorBlock:
=    0.0   98.9    89.1    0.0  block in executed code
                   89.1  100.0       UserProfile  >> dictionaryNames
-------------------------------------------------------

                   89.1  100.0       block in executed code
=    0.0   98.9    89.1    0.0  UserProfile  >> dictionaryNames
                   89.1  100.0       SymbolList   >> namesReport
-------------------------------------------------------

                   89.1  100.0       UserProfile  >> dictionaryNames
=    0.0   98.9    89.1    0.0  SymbolList   >> namesReport
                   89.1  100.0       SymbolList   >> names
-------------------------------------------------------

                   89.1  100.0       SymbolList   >> namesReport
=    0.0   98.9    89.1    0.0  SymbolList   >> names
                   89.1  100.0       AbstractDictionary >> associationsDetect:ifNone:
-------------------------------------------------------

                   89.1  100.0       SymbolList   >> names
=    0.0   98.9    89.1    0.0  AbstractDictionary >> associationsDetect:ifNone:
                   89.1  100.0       IdentityDictionary >> associationsDo:
-------------------------------------------------------
```

```
                    89.1  100.0       AbstractDictionary >> associationsDetect:ifNone:
=   23.2   98.9     89.1   23.4  IdentityDictionary >> associationsDo:
                     1.9    2.1       Object       >> _basicSize
                    27.5   30.9       block in AbstractDictionary >> associationsDetect:ifNone:
                    21.8   24.5       Array        >> _at:
                    17.1   19.1       AbstractDictionary >> _at:
-----------------------------------------------------

                    27.5  100.0       IdentityDictionary >> associationsDo:
=   11.6   30.5     27.5   37.9  block in AbstractDictionary >> associationsDetect:ifNone:
                    17.1   62.1       block in SymbolList >> names
-----------------------------------------------------

                    21.8  100.0       IdentityDictionary >> associationsDo:
=   24.2   24.2     21.8  100.0  Array        >> _at:
-----------------------------------------------------

                    17.1  100.0       block in AbstractDictionary >> associationsDetect:ifNone:
=   18.9   18.9     17.1  100.0  block in SymbolList >> names
-----------------------------------------------------

                    17.1  100.0       IdentityDictionary >> associationsDo:
=   18.9   18.9     17.1  100.0  AbstractDictionary >> _at:
-----------------------------------------------------

                     1.9  100.0       IdentityDictionary >> associationsDo:
=    2.1    2.1      1.9  100.0  Object       >> _basicSize
-----------------------------------------------------


================
STACK SAMPLING TREE RESULTS
elapsed CPU time:    90 ms
monitoring interval: 1.0 ms

 100.0% (95) executed code        [UndefinedObject]
   100.0% (95) ProfMonitor class    >> monitorBlock:downTo: [ProfMonitorTree class]
     100.0% (95) ProfMonitor          >> monitorBlock: [ProfMonitorTree]
       98.9% (94) block in executed code [ExecBlock0]
        |  98.9% (94) UserProfile          >> dictionaryNames
        |    98.9% (94) SymbolList           >> namesReport
        |      98.9% (94) SymbolList           >> names
        |        98.9% (94) AbstractDictionary >> associationsDetect:ifNone: [SymbolDictionary]
        |          98.9% (94) IdentityDictionary   >> associationsDo: [SymbolDictionary]
        |            30.5% (29) block in AbstractDictionary >> associationsDetect:ifNone: [ExecBlock1]
        |             |  18.9% (18) block in SymbolList  >> names [ExecBlock1]
        |            24.2% (23) Array                >> _at: [IdentityCollisionBucket]
        |            18.9% (18) AbstractDictionary   >> _at: [SymbolDictionary]
        |             2.1% (2) Object               >> _basicSize [IdentityCollisionBucket]
```

As you can see, the report is in four sections:

- STATISTICAL SAMPLING RESULTS
- STATISTICAL STACK SAMPLING RESULTS
- STATISTICAL METHOD SENDERS RESULTS
- STACK SAMPLING TREE RESULTS

Each section includes the same set of methods that the profile monitor encountered when it checked the execution stack every millisecond; the report is presented to give different views of this data.

Keep in mind that these numbers are based on sampling, and depending on the size and number of samples, may not exactly reflect the actual percentage of time spent in each method. If you may external calls to the OS, to user actions or other C libraries, this may also distort results for the invoking method.

If you enable object creation tracking, additional sections are included that report the count and object creation. For example:

Example 14.2 Object creation report

```
OBJECT CREATION REPORT:
elapsed CPU time:    40 ms
monitoring interval: 2.0 ms

tally  class of created object
           call stack
------  --------------------------------------

   600  String class
 - - -  - - - - - - - - - - - - - - - - - - - -
            500  SmallInteger >> asString
              500  SymbolList   >> namesReport
                500  UserProfile  >> dictionaryNames
                  500  executed code
                    500  GsNMethod class >> _gsReturnToC
 - - -  - - - - - - - - - - - - - - - - - - - -
            100  String class >> new
              100  SymbolList   >> namesReport
                100  UserProfile  >> dictionaryNames
                  100  executed code
                    100  GsNMethod class >> _gsReturnToC
------  --------------------------------------

   100  Array class
 - - -  - - - - - - - - - - - - - - - - - - - -
            100  SymbolList   >> names
              100  SymbolList   >> namesReport
                100  UserProfile  >> dictionaryNames
                  100  executed code
                    100  GsNMethod class >> _gsReturnToC
```

14.3 Modifying Cache Sizes for Better Performance

As code executes in GemStone, committed objects must be fetched from disk or from cache, and temporary objects must be managed. This is handled transparently by the GemStone repository monitor. The performance of your application can be affected both by the tuning of the caches, and the structure and usage patterns of your application.

GemStone Caches

GemStone uses four kinds of caches: temporary object space, the Gem private page cache, the Stone private page cache, and the shared page cache.

Two caches are associated with Gem processes: the temporary object space and the Gem private page cache. The other two caches (Stone private page cache and shared page cache) are associated with the Stone (although the Gem also makes use of the shared page cache).

Temporary Object Space

The *temporary object space* cache is used to store temporary objects created by your application. Each Gem session has a temporary object memory that is private to the Gem process and its corresponding session. When you fault persistent (committed) objects into your application, they are copied to temporary object memory.

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.

It is important to provide sufficient temporary object space. At the same time, you must design your application so that it does not create an infinite amount of reachable temporary objects. Temporary object memory must be large enough to accommodate the sum of live temporary objects and modified persistent objects. It that sum exceeds the allocated temporary object memory, the Gem can encounter an OutOfMemory condition and terminate.

The amount of memory allocated for temporary object space is primarily determined by the GEM_TEMPOBJ_CACHE_SIZE configuration option. You should increase this value for applications that create a large number of temporary objects — for example, applications that make heavy use of the reduced conflict classes or sessions performing a bulk load. (For more information about the reduced-conflict classes, see "Classes That Reduce the Chance of Conflict" on page 150.)

You will probably need to experiment somewhat before you determine the optimum size of the temporary object space for the application. The default of 10000 (10 MB) should be adequate for normal user sessions. For sessions that place a high demand on the temporary object cache, such as upgrade, you may wish to use 100000 (i.e., 100 MB).

For a more exhaustive discussion of the issues involved in managing the size of temporary object memory, and a general discussion of garbage collection, see the "Garbage Collection" chapter of the *System Administration Guide*.

For details about how to set the size of GEM_TEMPOBJ_CACHE_SIZE in the Gem configuration file, see the "GemStone Configuration Options" appendix of the *System Administration Guide*.

Gem Private Page Cache

The *Gem private page cache* is only used to hold bitmap pages and shadow object table pages during commit processing. When you commit objects created by your application, they move directly from temporary object memory to the shared page cache.

The amount of memory allocated for the Gem private page cache is determined by the GEM_PRIVATE_PAGE_CACHE_KB configuration option. The default size is 1000 KB; the minimum is 128 KB; the maximum is 524288 KB.

NOTE

Under normal circumstances, you should not need to modify the default values of the Gem private page cache.

Stone Private Page Cache

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 amount of memory allocated for the Stone private page cache is determined by the STN_PRIVATE_PAGE_CACHE_KB configuration option. The default size is 2000 KB; the minimum is 128 KB; the maximum is 524288 KB.

NOTE

Under normal circumstances, you should not need to modify the default values of the Stone private page cache.

Shared 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.

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.

For acceptable performance, the shared page cache should be large enough to hold the entire object table. To get the best possible performance, make the shared page cache as large as possible.

The amount of memory allocated for the shared page cache is determined by the SHR_PAGE_CACHE_SIZE_KB configuration parameter (in the Stone configuration file). The default size is 75000 KB; the minimum is 512 KB; the maximum is limited by the available system memory and the kernel configuration.

For details about how to set the size of SHR_PAGE_CACHE_SIZE_KB in the Stone configuration file, see the *System Administration Guide* (Appendix A, GemStone Configuration Options).

By default, only the system administrator is privileged to set this parameter, which is set at repository startup. 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.

Getting Rid of Non-Persistent Objects

As discussed in Chapter 4, you can create instances of KeySoftValueDictionary to enable your session to free up temporary object memory as needed. The entries in a KeySoftValueDictionary are *non-persistent*; that is, they cannot be committed to the database. When there is a demand on memory, you can configure GemStone to clear non-persistent entries as needed during a VM mark/sweep garbage collection.

The action taken during mark/sweep depends on two configuration parameters, along with *startingMemUsed* — the percentage of temporary object memory in-use at the beginning of the VM mark/sweep.

Case 1: GEM_SOFTREF_CLEANUP_PERCENT_MEM < *startingMemUsed* < 80%

If *startingMemUsed* is greater than GEM_SOFTREF_CLEANUP_PERCENT_MEM but less than 80%, the VM mark/sweep will attempt to clear an internally determined number of least recently used SoftReferences (non-persistent entries). Under rare circumstances, you might choose to specify a minimum number (GEM_KEEP_MIN_SOFTREFS) that will not be cleared.

Case 2: *startingMemUsed* < GEM_SOFTREF_CLEANUP_PERCENT_MEM

No SoftReferences will be cleared.

Case 3: *startingMemUsed* > 80%

VM mark/sweep will attempt to clear all SoftReferences.

For more about these and other configuration parameters, see the "GemStone Configuration Options" appendix of the *System Administration Guide*.

Several cache statistics may also be of interest: NumSoftRefsCleared, NumLiveSoftRefs, and NumNonNilSoftRefs. For more about these statistics, see the "Monitoring GemStone" chapter of the *System Administration Guide*.

14.4 Managing VM Memory

As mentioned earlier in this chapter, each Gem session has a temporary object memory that is private to the Gem process and its corresponding session. When you fault persistent (committed) objects into your application, they are copied to temporary object memory.

It is important to provide sufficient temporary object space. At the same time, you must design your application so that it does not create an infinite amount of reachable temporary objects. Temporary object memory must be large enough to accommodate the sum of live temporary objects and modified persistent objects. It that sum exceeds the allocated temporary object memory, the Gem can encounter an OutOfMemory condition and terminate.

There is a limit on how large a transaction can be, either in terms of the total size of previously committed objects that are modified, or of the total size of temporary objects that are transitively reachable from modified committed objects. For large applications, you may need to commit incrementally, rather than waiting to commit all at once.

The remainder of this chapter discusses issues to consider when allocating and managing temporary object memory, and presents techniques for diagnosing and addressing OutOfMemory conditions. This section assumes you have read the general discussion of memory organization in "Managing Memory" chapter of the *System Administration Guide*.

Large Working Set

If your application requires a large working set of committed objects in memory, you can configure the `pom` area to be large (compared to other object spaces) without having an adverse effect on in-memory garbage collection. To do this, increase the setting for the configuration parameter GEM_TEMPOBJ_POMGEN_SIZE. For details on how to do this, see the *System Administration Guide*, Appendix A.

Class Hierarchy

If your application references a very deep class hierarchy, you may need to adjust the memory configuration accordingly to allow a larger temporary object memory. When an object is in memory, its class is also faulted into the `perm` area of temporary object memory, along with the class's superclass, extending up through the hierarchy all the way to Object. While this approach provides for significantly faster message lookups, it also increases the consumption of temporary object memory.

For example, the default configuration provides 1 MB for the `perm` area. Each class consumes about 400 bytes (including the metaclass). Thus, the default configuration can accommodate about 2500 classes in memory at once.

UserAction Considerations

NOTE

Do not compact the `code` *region of temporary object memory while a UserAction is executing.*

When using GemBuilder for C, you may encounter an OutOfMemory error within an UserAction in either of the following situations:

- The UserAction faults in a large number of methods via **GciPerform**.
- The UserAction compiles a large number of anonymous methods via **GciExecute**.

Exported Set

The ExportSet is a collection of objects for which the Gem process has handed out its OOP to one of the interfaces (GCI, GBS, objects returned from topaz run commands). Objects in the ExportSet are prevented from being garbage collected by any of the garbage collection processes (that is, by a Gem's in-memory collection of temporary objects, or the epoch garbage collection). The ExportSet is used to guarantee referential integrity for objects only referenced by an application, that is, objects that have no references to them within the Gem.

The application program is responsible for timely removal of objects from the ExportSet. The contents of the ExportSet can be examined using hidden set methods defined in class System.

In general, the smaller the size of the ExportSet, the better the performance is likely to be. There are several reasons for this relationship. The ExportSet is one of the root sets used for garbage collection. The larger the ExportSet, the more likely it is that objects that would otherwise be considered garbage are being retained. One threshold for performance is when the size of the export set exceeds 16K objects. When its size is smaller than 16K objects, the export set is a small object in object memory. When its size is larger than 16K, the export set becomes a large object, implemented as a tree of small objects in memory.

The configuration parameter #GemDropCommittedExportedObjs will allow committed object to be removed from the ExportSet when memory is low, at the expense of having to re-fault these object when they are needed.

You can use GciReleaseObjs to remove objects from the ExportSet. For details, see the *GemStone/S 64 Bit GemBuilder for C* manual.

Debugging out of memory errors

If you find that your application is running out of temporary memory, you can set several GemStone environment variables to help you identify which parts of your application are triggering OutOfMemory conditions. These environment variables allow you to obtain multiple Smalltalk stack printouts and other useful information before your application runs out of temporary object memory. You can examine those printouts to determine how many objects of each class are in temporary memory. Once you've identified the cause/s of the problem, you can adjust your GemStone configuration options to provide the needed memory.

These environment variables are documented in the `$GEMSTONE/sys/gemnetdebug` file, which is a debug version of the `gemnetobject` script. They may be set for RPC processes using `gemnetdebug` in the gem login parameters, or via on the command line prior to starting linked topaz. For more information on these environment variables, see the *System Administration Guide*.

Signal on low memory condition

When a session runs low on temporary object memory, there are actions it can take to avoid running out of memory altogether; for example, the session may commit or abort, or discard temporary objects. By enabling handling for the notification `AlmostOutOfMemory`, an application can take appropriate action before memory is entirely full. This notification is asynchronous, so may be received at any time memory use is greater than the threshold the end of an in-memory markSweep. However, if the session is executing a user action, or is in index maintenance, the error is deferred and generated when execution returns.

After an `AlmostOutOfMemory` notification is delivered, the handling is automatically disabled. Handling must be reenabled each time the signal occurs. Handling this signal is enabled by executing either of the following:

```
System enableAlmostOutOfMemoryError
```

or

```
System signalAlmostOutOfMemoryThreshold: 0
```

When handling is enabled, the default threshold is 85%. You can find out the current threshold using:

```
System almostOutOfMemoryErrorThreshold
```

This will return -1 if handling is not enabled.

The threshold can be modified using:

`System Class >> signalAlmostOutOfMemoryThreshold:` *anInteger*
Controls the generation of an error when session's temporary object memory is almost full. Calling this method with 0 < *anInteger* < 100, sets the threshold to the given value and enables generation of the error.

Calling this method with an argument of -1 disables generation of the error and resets the threshold to the default.

Calling this method with an argument of 0 enables the generation of the error and does not change the threshold.

Methods for Computing Temporary Object Space

To find out how much space is left in the `old` area of temporary memory, the following methods in class System (category Performance Monitoring) are provided:

`System _tempObjSpaceUsed`
Returns the approximate number of bytes of temporary object memory being used to store objects.

`System _tempObjSpaceMax`
Returns the size of the `old` area of temporary object memory; that is, the approximate maximum number of bytes of temporary object memory that are usable for storing objects. When the `old` area fills up, the Gem process may terminate with an OutOfMemory error.

`System _tempObjSpacePercentUsed`
Returns the approximate percentage of temporary object memory that is being used to store temporary objects. This is equivalent to the expression:

```
(System _tempObjSpaceUsed * 100) //
System _tempObjSpaceMax.
```

Note that it is possible for the result to be slightly greater than 100%. Such a result indicates that temporary memory is almost completely full.

To measure the size of complex objects, you might create a known object graph containing typical instances of the classes in question, and then execute the following methods at various points in your *test* code to get memory usage information:

CAUTION
Do not execute this sequence in your production code!

Example 14.3

```
System _vmMarkSweep.
System _tempObjSpaceUsed.
```

Statistics for monitoring memory use

You can monitor the following statistics to better understand your application's memory usage. The statistics are grouped here with related statistics, rather than alphabetically.

Table 1 Statistics Related to the Objects Copied into Memory

ObjectsRead	The number of committed objects copied into VM memory since the start of the session.
ClassesRead	The number of classes copied into the `perm` generation area of VM memory since the start of the session.
MethodsRead	The number of GsNMethods copied into the `code` generation area of VM memory since the start of the session.
ObjectsRefreshed	The number of committed objects in VM memory that have been re-read from the shared page cache after transaction boundaries, since the start of the session.

Table 2 Statistics Related to Mark/Sweeps and Scavenges

NumberOfMark-Sweeps	The number of mark/sweeps executed by the in-memory garbage collector.
NumberOfScavenges	The number of scavenges executed by the in-memory garbage collector. Only updated at mark/sweeps.
TimeInMarkSweep	The real time (in milliseconds) spent in in-memory garbage collector mark/sweeps.
TimeInScavenge	The real time (in milliseconds) spent in in-memory garbage collector scavenges. Only updated at mark/sweeps.

Table 3 Statistics Related to Object Memory Regions

CodeCacheSizeBytes	Total size in bytes of copies of GsNMethods that are in the `code` generation area and ready for execution, as of the end of mark/sweep.
NewGenSizeBytes	The number of used bytes in the `new` generation at the end of mark/sweep.
OldGenSizeBytes	The number of used bytes in the old generation at the end of mark/sweep.

Table 3 Statistics Related to Object Memory Regions

PomGenSizeBytes	The number of used bytes in the pom generation area at the end of mark/sweep. Pom generation holds clean copies of committed objects.
PermGenSizeBytes	The number of used bytes in the perm generation area at the end of mark/sweep. Perm generation holds copies of Classes.
MeSpaceUsedBytes	The number of bytes occupied by the remembered set (remSet), in-memory oopMap, and in-use map entries.
MeSpaceAllocated-Bytes	The number of bytes allocated for the remembered set (remSet), in-memory oopMap, and map entries.

Table 4 Statistics Related to Stubbing

NumRefsStubbedMark-Sweep	The number of in-memory references that were stubbed (converted to a POM objectId) by in-memory mark/sweep.
NumRefsStubbedScav-enge	The number of in-memory references that were stubbed (converted to a POM objectId) by in-memory scavenge.

Table 5 Statistics Related to Garbage Collection

CodeGenGcCount	The number of times the code generation area has been garbage collected.
PomGenScavCount	The number of times scavenge has thrown away the oldest pom generation space.

Symbol Creation

In GemStone/S 64 Bit, a SymbolGem process runs in the background and is responsible for creating all new Symbols, based on session requests that are managed by the Stone. You can examine the following statistics to track the effect of symbol creation activity on temporary object memory.

Table 6 Statistics Related to Symbol Creation

NewSymbolRequests	The number of symbol creation requests by a session to the symbol creation gem.
NewSymbolsCount	The number of symbol creation requests by a session that did not resolve to an already committed symbol.
TimeWaitingForSym-bols	Cumulative elapsed time (in milliseconds) waiting for symbol creation requests to be processed.

Table 7 Other Statistics

ExportedSetSize	The number of objects in the ExportSet (see page 271).
TrackedSetSize	The number of objects in the Tracked Objects Set, as defined by the GCI. You can use GciReleaseObjs to remove objects from the Tracked Objects Set. For details, see the *GemStone/S 64 Bit GemBuilder for C* manual.
DirtyListSize	The number of modified committed objects in the temporary object memory dirty list.
WorkingSetSize	The number of objects in memory that have an objectId assigned to them; approximately the number of committed objects that have been faulted in plus the number that have been created and committed.
TempObjSpacePercentUsed	The approximate percentage of temporary object memory for this session that is being used to store temporary objects. If this value approaches or exceeds 100%, sessions will probably encounter an OutOfMemory error. This statistic is only updated at the end of a mark/sweep operation. Compare with `System _tempObjSpacePercentUsed` (page 273), which is computed whenever the primitive is executed.

14.5 NotTranloggedGlobals

All changes to the repository are written to the transaction logs when the transaction is committed, to ensure these changes are recoverable in case of unexpected shutdown, and to allow these changes to be applied to warm standby copies of the repository. However, you may have data that you will be committing changes to, but that does not need to be recovered in case of system crash or corruption. For this kind of data, you can avoid the overhead of writing each change to the transaction logs, and the disk space required for the transaction logs to archive large amounts of non-critical data.

For objects that are intended to be persistent, but not log changes in the transaction logs, there must be no reference from persistent objects, and the reference should be from the variable NotTranloggedGlobals. This is in the Globals SymbolDictionary.

For example:

```
NotTranloggedGlobals at: #perfLog put: PerformanceLogger new.
```

If the object in NotTranlogGlobals is reachable from AllUsers (the regular root for all persistent objects), it will generate an error on commit.

On system crash or unexpected shutdown, the state of the objects reachable from NotTranloggedGlobals will be as was recorded in the most recent checkpoint prior to the shutdown; changes made after that checkpoint will be lost. If the repository is restored

from backup, and transaction logs applied, the state of these objects will be as of the time the backup was taken; all changes made since the backup was taken are lost.

14.6 Other 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.
- The following Number classes are listed in decreasing order of performance:

  SmallInteger
  SmallDouble
  Float
  LargeInteger
  ScaledDecimal
  DecimalFloat

- Avoid coercing integers to floating point numbers. Although GemStone 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. (For information on writing your own primitive methods, see the *GemBuilder for C* manual.)
- Prefer message-sends over path notation, where possible. (This is not possible in indexed queries, however.)
- 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.

  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.
- Used optimized selectors whenever possible. For example, iterations using `to:do` are specially optimized; using `to:do:` instead of another collection iteration method avoids a message send and a level of block nesting, possibly avoiding the cost of using a block altogether. See page 331 for a list of optimized selectors.

In the same way, for fastest performance in iterating over Collections, use the `to:do:` or `to:by:do:` methods to iterate, rather than `do:` or other collection iteration methods

- Resize rather than concatenate strings. `String >> ,` creates a new string to use in modifying the old one, whereas `String >> add:` modifies a string. 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.
- Avoid generating temporary objects whose storage will need to be reclaimed. Storage reclamation can slow your application significantly.
- Keep repository files on a disk reserved for their use, if possible. Particularly avoid putting repository files on the disk used for swapping.
- For large applications, you may need to commit incrementally, rather than waiting to commit all at once. There is a limit on how large a transaction can be, either in terms of the total size of previously committed objects that are modified, or of the total size of temporary objects that are transitively reachable from modified committed objects.
- Consider trade-offs in indexing. While indexes can improve query performance on large collections, there is overhead. If the collection has fewer than about 2000 objects, the extra overhead in internal objects and index maintenance may not be worth negligable performance gain in queries.

Chapter

15 Working with Classes and Methods

An object responds to messages defined and stored with 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.

This chapter provides an overview of the Behavior methods that are inherited by all classes, and so can be used to programmatically create and access methods, categories, pool dictionaries and variables for your classes.

Creating and Removing Methods
describes the protocol in class Behavior for adding and removing methods.

Information about Class and Methods
describes the protocol in class Behavior for examining the method dictionary of a class.

ClassOrganizer
describes the protocol in class Behavior for examining, adding, and removing method categories.

Handling Deprecated Methods
How to locate and clean up references to methods that have been deprecated.

15.1 Creating and Removing Methods

Class Behavior defines messages for creating methods and removing methods.

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 another method that returns it. These methods are added to the categories "Accessing" (return the instance variable's value) and "Updating" (set its value).

For example, this invocation of the method:

```
Animal compileAccessingMethodsFor: #(#name)
```

has the same effect as the following topaz:

```
category: 'Accessing'
method: Animal
name
    ^name
%
category: 'Updating'
method: Animal
name: aName
    name := aName
%
```

You can also use `compileAccessingMethodsFor:` to define class methods for accessing class, class instance and pool variables, by sending `compileAccessingMethodsFor:` to the *class* of the class that defines the variables of interest.

The similar method `compileMissingAccessingMethods` will create accessing methods for any instance variables for which accessor methods with the standard selector do not already exist.

Compiling Methods

Class Behavior defines the basic method for compiling a new method for a class and adding the method to the class's method dictionary.

An invocation of the method has this form:

```
aClass compileMethod: sourceString
       dictionaries:  arrayOfSymbolDicts
       category:      aCategoryNameString
       environmentId: 0
```

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 third argument names the category to which the new method is to be added.

environmentId specifies one of potentially multiple compile environments, provided for Ruby implementations; it is normally 0 for Smalltalk applications. You can omit this keyword, and methods within Smalltalk will default to an environmentId of 0.

The following code compiles an accessor method named `habitat` for the class Animal, adding it to the category "Accessing":

```
Animal
   compileMethod:
        'habitat
         "Return the value of the receiver''s habitat
          instance variable"
         ^habitat'
   dictionaries: (System myUserProfile symbolList)
   category: 'Accessing'
   environmentId: 0
```

When you write methods for compilation in this way, remember to double each apostrophe within the source string.

If `compileMethod:..` executes successfully, it adds the new method to the receiver. If the source string contains errors, this method signals a CompileError, with details on the specific causes of the failure.

Removing Methods

You can remove a method by sending `removeSelector:` *aSelectorSymbol* to a class or metaclass.

The following examples remove instance and class methods, respectively:

```
Animal removeSelector: #habitat

Animal class removeSelector:#newWithName:favoriteFood:habitat:
```

To remove all methods in a method category, as well as the category itself, use `removeCategory:` *categoryName*. For example,

```
Animal removeCategory: 'Accessing'
```

15.2 Information about Class and Methods

Classes Behavior and Class defines messages that let you discover information about a class, such as the class's instance variables, selectors, and categories. The class ClassOrganizer provides searching over methods in the image.

For full protocol, see the image.

Information about the Class

Protocol in Class provides listing of superclasses and subclasses:

```
Class >> allSubclasses
Class >> allSuperclasses
Class >> allInstances
```

Each class also has a class comment and a category. This information can be accessed and updated using:

```
Class >> comment
Class >> comment: aString
Class >> category
Class >> category: aString
```

Information about Instance, Class, and Shared Pool variables

Protocol in Behavior allows you to discover the class variables names, instance variable names, and shared pools defined for a given class, or for that class and all its superclasses.

```
Behavior >> classVarNames
Behavior >> allClassVarNames
Behavior >> instVarNames
Behavior >> allInstVarNames
Behavior >> sharedPools
Behavior >> allSharedPools
```

Information about Method Selectors

Protocol in Behavior allows you to discover the selectors for the methods in a class, or in that class and its superclasses, and query on particular selectors.

```
Behavior >> selectors
Behavior >> allSelectors
Behavior >> includesSelector: aSelector
Behavior >> canUnderstand: aSelector
Behavior >> whichClassIncludesSelector: aSelector
```

Accessing and Managing Method Categories

The methods in a class are associated with a method category, which is used to organize and document the method but does not affect execution. Method categories can be managed programmatically using the following methods in Behavior:

```
Behavior >> categoryNames
Behavior >> selectorsIn: categoryName
Behavior >> categoryOfSelector: selector
Behavior >> addCategory: categoryName
Behavior >> removeCategory: categoryName
Behavior >> renameCategory: categoryName to: newCategoryName
Behavior >> moveMethod: aSelector toCategory: categoryName
```

Specific Methods

Each method is compiled into an instance of GsNMethod. You can query a class for its methods, and get source code and other information about the method.

To get the source code for a method, use:

```
Behavior >> sourceCodeAt: aSelector
```

To retrieve the compiled method itself, use:

```
Behavior >> compiledMethodAt: aSelector
```

This returns an instance of GsNMethod, from which you can then get source code. For example,

```
(Animal compiledMethodAt: #habitat) sourceString
```

Some GsNMethod methods that may be particularly useful are:

```
GsNMethod >> sourceString
GsNMethod >> sourceStringToFirstComment
GsNMethod >> selector
```

15.3 ClassOrganizer

ClassOrganizer provides useful methods to analyze your repository and perform operations such as searching for senders, receivers, or implementors, and string searches over method source. While usually you would perform these operations using GBS (or another Smalltalk IDE), ClassOrganizer provide the ability to do customized analysis and reporting.

ClassOrganizer provides both reporting methods, which return formatted Strings, and query methods, which return collections of symbols or instances of GsNMethods that can be used for further analysis and reporting.

For example, to get a report of all the senders of #asDecimalFloat:

```
ClassOrganizer new sendersOfReport: #asDecimalFloat
%
DecimalFloat >> integerPart
DecimalFloat >> rem:
DecimalFloat >>  coerce:
FixedPoint >> asDecimalFloat
Fraction >> asDecimalFloat
ScaledDecimal >> asDecimalFloat
SmallFloat >> asDecimalFloat
```

If you want to perform more analysis on the methods or add additional reporting, send `sendersOf:`, which will return two arrays, the first an array of GsNMethods, the second the offset into the source code. For example

```
(ClassOrganizer new sendersOf: #asDecimalFloat) printString
%
anArray( anArray( aGsNMethod, aGsNMethod, aGsNMethod,
aGsNMethod, aGsNMethod, aGsNMethod, aGsNMethod), anArray( 161,
102, 309, 104, 215, 1052, 85))
```

See the image for the full set of protocol that ClassOrganizer understands.

For example, the following code looks for all methods that are send the message `subclassRresponsibility:`, and make sure all subclasses override that implementation. This example will return false positives, however, since it does not distinguish abstract classes.

```
| clsOrg meths report |
clsOrg := ClassOrganizer new.
report := String new.
meths := (clsOrg sendersOf: #subclassResponsibility:) at: 1.
meths do:
   [:srMeth |
   (clsOrg subclassesOf: srMeth inClass) do:
      [:subcls |
      (subcls whichClassIncludesSelector: srMeth selector) =
            srMeth inClass
         ifTrue: [
            report
               add: subcls name asString;
               add: ' does not override ';
               add: srMeth inClass asString;
               add: '>>';
               add: srMeth selector asString;
               lf
               ].
      ]
   ].
report
```

15.4 Handling Deprecated Methods

As GemStone features change, some methods may no longer be appropriate, or the method names may be incorrect or misleading. To allow obsolete methods to continue to function and provide a gentle transition to new methods, these obsolete methods may be deprecated.

Deprecated methods may be removed in future major releases, although some deprecated methods may remain in the image for longer periods for the convenience of existing applications.

Usually, deprecated methods will continue to work exactly as they did in the previous releases. However, in some cases the old behavior may not be meaningful in a new version; the deprecated method will continue to work as similarly as possible, but there may be differences.

Behavior may also change for existing methods. With any new release, you should review the Release Notes for changes in behavior as well as for newly deprecated methods.

Deprecated methods in GemStone are indicated by:

- Officially deprecated method must include a call to `deprecated:`.
- Deprecated methods are in method category with a name including 'Deprecated'.
- Deprecation may be mentioned in the method comment. This may indicate an intention to deprecate.

Private methods, in a category with a name including 'Private', or which begin with an underscore, or which the method comment says private, may or may not be deprecated prior to removal. It is strongly recommended to avoid calling private methods.

Kernel methods that call `deprecated:` provide a string, which will generally include the class and selector, the version in which this method was deprecated, and the method that replaces it or some other indications of alternate action.

Since deprecated methods are subject to removal in major releases, it is important to keep your application updated so that no deprecated methods are called.

Deprecated handling

By default, nothing happens when a deprecated method is called; the call to deprecated: has no action. This is most convenient when you first upgrade or convert to a new release of GemStone.

After you have updated your application references to deprecated methods, you can enable Deprecation handling, which can be configured to error or to log all calls to any deprecated methods. By running with this setting, you can locate and fix calls you may h ave missed, or confirm that you have indeed fixed all calls.

Changing deprecation handling can only be done by a user with write permission for the DataCurator object security policy. Once committed, the setting affects all users of the repository.

There are several levels of action that can be taken when a deprecated method is called:

- **Do nothing** -- calls to deprecated methods are execute the same as any other method. This is the default.

To turn off any action on deprecation that you have previously enabled, execute:

```
Deprecated doNothingOnDeprecated
```

- **Raise an exception** -- calls to deprecated methods signal an exception.

  To enable this, execute:

  ```
  Deprecated doErrorOnDeprecated
  ```

- **Log the call** -- when a call to a deprecated method occurs, the call to the deprecated method is logged to the deprecation log file, and execution continues. There is no impact on the application, other than performance.

  To enable this, execute:

  ```
  Deprecated doLogOnDeprecated
  ```

- **Log the call stack** --when a call to a deprecated method occurs, the call to the deprecated method and the call stack are logged to the deprecation log file, and execution continues. There is no impact on the application, other than performance.

  To enable this, execute:

  ```
  Deprecated doLogStackOnDeprecated
  ```

Deprecation log

When deprecations are configured to write to a log, a file named Deprecated*PID*.log is created in the same location as a the gem log for an RPC login.

This file continues to grow and must be manually deleted. Logging methods or call stacks consumes resources and can noticeably affect performance, and use significant disk space. Methods called repeatedly, such as calls from within sort blocks, are particularly likely to impact the application.

Listing deprecated methods

You can find all currently deprecated methods in a particular version by executing :

```
ClassOrganizer new sendersOfReport: #deprecated:
```

Determining senders of deprecated methods

For each deprecated method, you can use development tools to determine if you have any senders within your application. In addition to GBS or other IDE tools, you can use ClassOrganizer methods.

For example, having determined that `setSegmentId:` has been deprecated, you can perform this to find all senders of that selector within your application:

```
ClassOrganizer sendersOfReport: #setSegmentId:
```

Since deprecation only applies to a method associated specific class, and this search looks for all senders of the selector, you will have to examine the list to determine if the call is actually deprecated. This is the consequence of how typing is handled in Smalltalk. For example, String >>+ is deprecated, but Integer >>+ is not.

This will not find methods in perform: statements, in code executed by client applications, or in topaz scripts.

Chapter

16 System Sets

GemStone provides an interface to number of internal structures that provide specialized behavior, different from the way normal objects are handled with respect to storage, visibility to other sessions, and transactional behavior. These structures are intended for use by experienced GemStone programmers.

This chapter provides an introduction to these specialized structures.

Hidden Sets
Describes HiddenSets, a non-persistent way to manage objects using bitmaps.

SessionTemps and access to Session State
Ways to keep session-temporary data available for the life of a session.

Shared Counters
Integer counters that can be shared between sessions. Both non-persistent and persistent counters are available.

16.1 Hidden Sets

Hidden sets are internal GemStone structures that are used to hold objects in the form of OOPs. They are implemented as bitmaps, an efficient way to transfer large collections of objects. Hidden sets use heap memory, not temporary object cache memory, and the objects in the hidden set are not loaded in memory, so hidden sets can be very useful when working with very large collections.

Several repository-wide operations, such as `listInstancesToHiddenSet:`, write the results to a hidden set; this allows operations that may return very large result sets to complete, and the results to be enumerated, without exceeding memory limits.

Many hidden sets are used internally, but there are a number of hidden sets that are provided for customer use. The specific hidden sets and their purposes are documented in the image method `System class >> HiddenSetSpecifiers`. Hidden sets number 41 through 45 are designated for use by customers for their applications.

Hidden sets are ordered in OOP order. You can load hidden sets from data that is organized in any order, such as files containing oops sorted in page order, but that ordering will be lost in the hidden set. Any OOP can only appear once in a hidden set; so, like an IdentitySet, identical objects can only appear once, but equal objects can both be included.

Special objects that are encoded within the OOP cannot be stored in Hidden sets. Attempting to add objects such as SmallIntegers, SmallDoubles, Characters and Booleans to a Hidden set will result in an error.

For example, the method `listInstancesToHiddenSet:` puts the results of a listInstances operation in Hidden Set 1. The following code shows the call to this method, and how to use hidden set protocol migrate each object:

Example 16.1

```
topaz 1> run
SystemRepository listInstancesToHiddenSet: MyClass.
[(System hiddenSetSize: 1) > 0]
   whileTrue:
   [ | resultBatch |
   resultBatch := System hiddenSetEnumerate: 1 limit: 1024.
   resultBatch do: [:aMyClass |
         aMyClass migrate].
   System commitTransaction.
     ].
%
```

Methods to work with Hidden Sets

Add

You can add a single object or an array of objects to a hidden set using the methods:

`System Class >> add:` *anObject* `toHiddenSet:` *hiddenSetSpecifier*

`System Class >> addAll:` *anArray* `toHiddenSet:` *hiddenSetSpecifier*

To add all objects in one hidden set to another hidden set, use:

`System class >> addHiddenSet:` *hiddenSet1* `to:` *hiddenSet2*

Remove

You can remove a single object or an array of objects from a hidden set using the methods:

`System Class >> remove:` *anObject* `fromHiddenSet:` *hiddenSetSpecifier*

`System Class >> removeFirst:` *count*
`fromHiddenSet:`*hiddenSetSpecifier*

`System Class >> removeAll:` *anArray*
`fromHiddenSet:`*hiddenSetSpecifier*

```
System Class >> removeContentsOfHiddenSet: hiddenSet1
   fromHiddenSet: hiddenSet2

System Class >> truncateHiddenSet: hiddenSetSpecifier
   toSize:newSize
```

Any objects to be removed that are not in the hidden set are ignored. For more details, see the method comments in the image.

To reinitialize the hidden set, removing all objects, use the following:

```
System Class >> hiddenSetReinit: hiddenSetSpecifier
```

Testing

To determine how large the hidden set is, use the method:

```
System Class >> hiddenSetSize: hiddenSetSpecifier
```

To determine if a specific object is in the hidden set, use:

```
System Class >> testIf: anObject isInHiddenSet: hiddenSetSpecifier
```

Set operations

To compute the union or difference of two hidden sets, and place the results in a third hidden set, use the following methods:

```
System Class >> computeUnionOfHiddenSet: hiddenSet1 and:
   hiddenSet2 into: hiddenSet3
System Class >> computeDifferenceOfHiddenSet: hiddenSet1
   and: hiddenSet2 into: hiddenSet3
```

Enumerating

Retrieving the contents of hidden sets is done through the following methods. These methods return a chunk of the contents of the hidden set as objects or as OOPs. These objects are removed from the hidden set.You can then perform whatever operations you need on each object in this chunk, before fetching another chunk. This way, very large collections of objects can be operated on.

```
System Class >> hiddenSetEnumerate: hiddenSetSpecifier
   limit: maxResultSize
```

This method returns the first *maxResultSize* objects in the hidden set. If there are not that many objects in the hidden set, the result may be smaller than *maxResultSize*. If *maxResultSize* is 0, all objects are returned (similar to `hiddenSetAsArray:`).

```
System Class >> hiddenSetEnumerateAsInts: hiddenSetSpecifier
   limit: maxResultSize
```

This method is the same as `hiddenSetEnumerate:limit:`, except the OOPs of the objects are returned, rather than the objects.

Converting

To create an Array containing all objects in the hidden set, use the following method.

```
System Class >> hiddenSetAsArray: hiddenSetSpecifier
```

Some care should be taken not to use this with very large hidden sets. The objects in the resulting array, unlike the objects in the hidden set, are in temporary object memory. If the hidden set is too large it may cause the session to run out of memory.

16.2 SessionTemps and access to Session State

Most data that you will work with in GemStone is either temporary or persistent. While most temporary data is only retained for as long as the method is executing, or until the session updates its commit record by committing or aborting, you may sometimes want data that is not persistent and not shared, so does not risk transaction conflicts, but remains unaffected by transaction status.

Session-specific data of this kind can be put into SessionTemps. `SessionTemps current` provides access to a kind of SymbolDictionary; elements in the SessionTemps dictionary remain until the session logs out or exits, are not affected by commit or abort, and are not visible outside of the session.

For example, if you wish to open a log file and leave it open:

```
SessionTemps at: #Log put: (GsFile openAppend: 'myFile.log')
```

Actual code, of course, would do more error checking. To write to the file, use code similar to this:

```
(SessionTemps at: #Log) nextPutAll: 'a message for the log
file'.
```

Objects in SessionTemps use temporary object memory, and the objects cannot be removed from memory by in-memory garbage collection. While there is no limit on how much data can be stored in SessionTemps, if your session reaches the memory limit and exits, that data will be lost.

SessionState

SessionTemps uses a slot in the internal Session State structure, which is primarily provided for use by the kernel. Access to customer-available SessionState slots is provided primarily for legacy uses, but may be useful depending on application requirements.

SessionState is accessed by integer index, with slots 1 to 1994 available for use. The SessionState array is variable size, and will grow as needed.

The following methods can be used to read and update SessionState:

```
System >> sessionStateAt: anIndex
System >> sessionStateAt: anIndex put: anObject
System >> sessionStateSize
```

16.3 Shared Counters

There are two types of Shared Counters available; AppStat Shared Counters and Persistent Shared Counters.

AppStat Shared Counters provide a way for sessions on the same shared cache to read and update a set of counters. These counters are stored in the shared cache and are not persistent across cache restart. They are not visible to sessions on remote shared page caches, nor are the values recoverable from tranlogs.

Persistent shared counters are stored in the repository, and are visible to all sessions on all shared caches. On repository recovery or restore, the values of persistent shared caches are restored.

AppStat Shared Counters

Shared counters allow multiple sessions on the same SPC to read and update a common counter value.

Shared counters are indexed from 0 to (System numSharedCounters - 1), which is set by the configuration parameter SHR_PAGE_CACHE_NUM_SHARED_COUNTERS. The default value for SHR_PAGE_CACHE_NUM_SHARED_COUNTERS is 1900. Each counter is protected by a unique spinlock. The index of the first counter is 0.

Shared counters may be set to any signed 64 bit integer value, in the range:
-2^{63} (-9223372036854775808) to 2^{63} - 1 (9223372036854775807)
If you increment or decrement so that the result would be outside the range of a signed 64-bit integer, the value will be set to the minimum or maximum; directly setting an out of range value will result in an error.

Shared counters are transient, that is, they do not persist across cache restart.

Shared counter values are recorded by statmonitor when using the -n option and recorded as AppStats.

The following methods may be used to read and update shared counters. For details, see the method comments in the image.

```
System class >> numSharedCounters
System class >> sharedCounter: index
System class >> sharedCounter: index setValue: value
System class >> sharedCounter: index incrementBy: amount
System class >> sharedCounter: index decrementBy: amount
System class >> sharedCounter: index decrementBy: amount
   withFloor: floorValue
sSystem class >> sharedCounterFetchValuesFrom: firstCounter
   to: lastCounter
```

Persistent Shared Counters

Persistent shared counters allow all sessions in a repository to read and update a set of counters. Persistent shared counters are globally visible to all sessions on all shared page caches.

There are 1536 persistent shared counters, numbered from 1 to 1536. The index of the first counter is 1.

Persistent shared counters may be set to any signed 64 bit integer value, in the range:
-2^{63} (-9223372036854775808) to 2^{63} - 1 (9223372036854775807)
No limit checks are done when incrementing or decrementing a counter. If you increment or decrement so that the result would be outside the range of a signed 64-bit integer, the value will "rollover" and the overflow bits will be lost. Directly setting an out of range value will result in an error.

Values of all persistent shared counters are stored in the repository and in tranlog records. They are persistent through Stone restart, and recovered on Stone crash, restore from backup, and restore from tranlog.

Persistent shared counters are independent of database transactions. Updates to counters are visible immediately and not affected by aborts.

Each update to a persistent shared counter causes a roundtrip to the Stone; but reading the value is handled by the gem (and the page server, if remote), and does not cause a roundtrip to the stone.

The following methods may be used to read and update persistent shared counters. For details, see the method comments in the image.

```
System class >> numberOfPersistentSharedCounters
System class >> persistentCounterAt: index put: value
System class >> persistentCounterAt: index
System class >> persistentCounterAt: index incrementBy: amount
System class >> persistentCounterAt: index decrementBy: amount
```

Chapter

17 The Foreign Function Interface

For certain applications, you may need to provide functionality that is not readily available within GemStone Smalltalk. Such functionality might include interactions with third-party products such as these:

- Access to hardware, such as a bar code reader
- Access to software that provides a service, such as the zlib compression library
- Data encryption
- Screen graphics
- Interaction with Oracle, mySQL, or other databases

To interact with third-party products such as these, you can use the Foreign Function Interface (FFI) to make C library calls from within GemStone Smalltalk. Using the FFI, you can access C functions in external libraries without the need to write UserActions.

NOTE

With UserActions, your code is checked against function prototypes of the external library that you're calling. With the FFI, no such checking takes place.

This chapter describes the FFI classes and methods, and how you can use them to build and interface to an existing C library..

FFI Core Classes
describes the FFI related classes and data types.

FFI Wrapper Utilities
Instructions for using FFI utilities to define FFI classes for your library.

17.1 FFI Core Classes

The core FFI defines six classes: CLibrary, CFunction, CPointer, CByteArray, CCallout, and CCallin.

CLibrary

An instance of CLibrary corresponds to a C compiled library. Instances of CLibrary are created using:

```
CLibrary class >> named:libraryName
```

passing in the path and name of the C shared library to be loaded. The platform-specific extension (such as .so) is optional.

CCallout

Individual functions within a CLibrary are represented by instances of CCallout. To create a CCallout, the following class methods are available:

`library:` *aCLibrary* `name:` *aName* `result:` *resType* `args:` *argumentTypes*

`library:` *aCLibrary* `name:` *aName* `result:` *resType* `args:` *argumentTypes* `varArgsAfter:` *varArgsAfter*

`name:` *aName* `result:` *resType* `args:` *argumentTypes*

`name:` *aName* `result:` *resType* `args:` *argumentTypes* `varArgsAfter:` *varArgsAfter*

aCLibrary may be an instance of CLibrary, an Array of CLibraries, or nil. Passing nil for *aCLibrary* will cause search of the loaded libraries for a function of this name. *aName* is a String providing the name of the specific function. *resType* is the return type of the function, and *argumentTypes* is an array of zero or more symbols describing the types of the argument for this function.

varArgsAfter is -1 if the number of arguments to the function is fixed. If the function prototype ends with an ellipsis ('...'), indicating that the function takes a variable number of arguments, then *varArgsAfter* indicates the one-based index of the last fixed argument. (If *varArgsAfter* is 0, there are no fixed arguments.)

The following instance method is used to invoke the function described by the instance of CCallout:

`callWith:` *argsArray*

To get the value of the C global variable errno that was saved by the most recent call to callWith:, use the CCallout class method:

```
errno
```

C type symbols

Table 1 lists the symbols used for creating *resType* (result type) and *argumentTypes* arguments when creating CCallouts.

Table 1 C type symbols

	Return type	Argument type
#int64	Integer. The C function returns an int64.	Integer
#uint64	Integer. The C function returns a uint64.	Integer
#int32	Integer. The C function returns a signed C integer 32 bits.	Integer
#uint32	Integer. The C function returns an unsigned C integer, 32 bits or smaller.	Integer
#int16	Integer	Integer
#uint16	Integer	Integer
#int8	Integer	Integer
#uint8	Integer	Integer
#double	SmallDouble or Float. The C function returns a C double.	SmallDouble or Float; and the function is limited to a maximum of four arguments.
#float	SmallDouble or Float. The C function returns a C float.	SmallDouble or Float
#'char*'	nil or a String	The corresponding arg must be a String. The body is copied to C memory before call and copied from C memory (and possible grown/shrunk) after call. C memory will not be valid after the call finishes.
#void	nil	
#ptr	nil or a CPointer	The corresponding arg must be nil, a CByteArray or a CPointer. If nil, a C NULL is passed. If CByteArray, address of body is passed. If CPointer, the encapsulated pointer is passed.
#'&ptr'		The corresponding arg must be a CPointer. The CPointer's value will be passed and updated on return.

Table 1 C type symbols (Continued)

	Return type	Argument type
#'&int64'		The corresponding arg must be a CByteArray of size 8. A pointer to body will be passed.
#'&double'		The corresponding arg must be a CByteArray of size 8. A pointer to body will be passed.
#'const char*'		The corresponding arg must be nil (to pass NULL) or a String (body is copied to C memory before call) C memory will not be valid after the call finishes.

Functions using varArgs normally may have a maximum of 20 variable arguments. This limit is lower if native code is disabled for this session; see "Limitations with native code disabled" on page 297.

Limitations with native code disabled

If the generation of native code is disabled, there are further limitations:

- Functions using varArgs may have a maximum of four fixed and 10 total arguments.
- Functions not using varArgs are limited to a maximum of 15 total arguments.
- Arguments and results of C type float are not supported.
- Functions with one or more args of C type double are limited to a maximum of four arguments.
- CCallin cannot be used

Native code generation is on by default, but may be configured to be disabled or becomes disabled when breakpoints are set. See the *System Administration Guide* for more information on native code generation.

CCallin

A CCallin represents a signature for a C function to be called by C code. The resulting CCallin may be used as a type within the *argumentTypes* array when defining a CCallout.

CByteArray

A CByteArray represents an allocation of C memory. When objects such as pointers or strings are passed to or from C functions, creating a CByteArray, with memory malloc'ed, ensures that the memory will be valid following the call.

CFunction

CFunction is an abstract superclass representing the type signature of a C function. It has two subclasses, CCallout and CCallin.

CPointer

CPointer encapsulates a C pointer that does not have auto-free semantics. New instances are created by CFunction calls with result type #ptr, and are also used for certain arguments of CFunctions.

17.2 FFI Wrapper Utilities

While it is possible to manually construct FFI calls using the core classes described above in section 17.1, it involves analysis of the various header files and may be tedious and error-prone. The typical header file includes many other header files, and the typical C program involves many defines, typedefs, and other definitions.

To help in the process of constructing FFI calls, GemStone includes a class, CHeader, that does the required analysis of a header file. You can parse a header file by using the method `CHeader class >> path:`. This will return an object containing an analysis of the header file.

The following example analyzes a a header file and stores the result in a variable in UserGlobals:

Example 17.1 Create a CHeader for zlib.h

```
topaz 1> doit
UserGlobals at: #'ZLibHeader' put:
   (CHeader path: '/usr/include/zlib.h').
%
```

NOTE

Many of the following examples use zlib, a software library for data compression that is available on many platforms. Documentation on the library is available at http://zlib.net/manual.html. *These zlib examples are on Linux; library details are platform-specific. If you are trying these examples on another platform, you may need to experiment.*

Once you have a CHeader object, you can get information about the various things defined in the header file and those it includes.

Example 17.2 CDeclaration for compress()

```
topaz 1> printit
(ZLibHeader functions at: 'compress')
%
a CDeclaration
  header                a CHeader
  name                  compress
  storage               extern
  type                  int32
  count                 nil
  pointer               0
  fields                nil
  parameters            a Array
  enumTag               nil
  isStorage             false
  isConstant            false
  includesCode          false
  isVaryingArgCount false
  isTransparentUnion false
  bitCount              nil
  source               \n/* Return flags indicating compile-time
options.\n\n    Type ...
  file                  /usr/include/zlib.h
  line                  1042
```

While the `compress()` function is directly in `zlib.h`, this isn't necessarily the case. Functions that are defined in any header file that is #included in the parsed header file also will have definitions in the instance of CHeader.

For example, on Linux the `zlib.h` file #includes `unistd.h`, so functions such as `getcwd()` also have definitions in the instance of CHeader:.

```
topaz 1> run
(ZLibHeader functions at: 'getcwd') file.
%
/usr/include/unistd.h
```

On other platforms, `zlib.h` may not #include `unistd.h`. In this case, the definition is not included in ZLibHeader. In this case (if you wanted to access these functions from GemStone), you could create a separate instance of CHeader for `unistd.h`:

```
topaz 1> doit
UserGlobals at: #'UnistdLibHeader' put: (CHeader path:
'/usr/include/unistd.h').
```

Note that parsing the header file does not give you the location of the actual C library file that you will be calling. Normally when to write an interface to specific libraries, you would be provided the library names and locations as well as the header files.

Simple function call--getcwd()

To take an example that is in `unitstd.c`, viewing the source for the `getcwd()` function declaration will let us see the argument declarations.

```
topaz 1> run
(ZLibHeader functions at: 'getcwd') source
%
/* Get the pathname of the current working directory,
   and put it in SIZE bytes of BUF.  Returns NULL if the
   directory couldn't be determined or SIZE was too small.
   If successful, returns BUF.  In GNU, if BUF is NULL,
   an array is allocated with `malloc'; the array is SIZE
   bytes long, unless SIZE == 0, in which case it is as
   big as necessary.  */
extern char *getcwd (char *__buf, size_t __size) __THROW __wur;
```

This tells us that the function takes two arguments, a pointer to a string and an integer, and returns a pointer to a string. Knowing that the function defined by this header is in `libc`, and the actual library path and filename is `/lib/libc.so.6`, we can manually create a call to this function:

Example 17.3 CCallout to invoke getcwd()

```
| string ccallout_getcwd |
string := String new: 200.
ccallout_getcwd := CCallout
   library: (CLibrary named: '/lib/libc.so.6')
   name: 'getcwd'
   result: #'char*'
   args: #(#'char*' #'uint64').
string := ccallout_getcwd callWith:
   (Array with: string with: string size).
```

It's important to note the way arguments are defined, since C handles memory differently from Smalltalk. The temporary string that is created as an argument to the function must be created with a size larger than the expected result. This is required for heap space to be allocated for the C function; if it is not large enough, the function will error. Also keep in mind that it's very important that the specified size of the string in the second argument not be larger than the actual size of the string. The C function will write results to memory limited by the second argument.

`getcwd()` updates the argument as well returns a value; both contain the same string, but different instances. In both cases String's size is now the actual size of the returned String, truncated from the original size of 200.

More complex function call--compress()

A more complex example is the ZLib function `compress()`. This is defined in `zlib.h` as follows:

```
ZEXTERN int ZEXPORT compress OF((Bytef *dest, uLongf *destLen,
const Bytef *source, uLong sourceLen));
```

You can view a simplified definition using the CHeader printString:

```
topaz 1> printit
(ZLibHeader functions at: 'compress') printString
%
extern int32 compress(uint8 *dest, uint64 *destLen, uint8
*source, uint64 sourceLen)
```

This tells us that `compress()` takes four arguments:

- a pointer to a destination buffer
- a pointer to the length of the destination buffer
- a pointer to the source data
- the length of the source data

The function compresses the source data and places the result in the destination buffer. The destination length is updated with the space actually used. The function returns a flag indicating success or the type of error experienced.

We can manually create a call to this function using the core classes described in 16.1:

```
CCallout
   library: (CLibrary named: '/lib/libz.so.1.2.3.3')
   name: 'compress'
   result: #'int32'
   args: #(#'ptr' #'ptr' #'const char*' #'uint64').
```

This creates an object that can be used to call the compress() function in the library. The constructor takes four arguments: (1) an instance of CLibrary; (2) the name of the function; (3) the result type; and (4) a list of the types of the arguments.

In order to call the function from Smalltalk we need to create the arguments. The source string and the source length are easy--they are just instances of a Smalltalk String and Integer. The destination and destination length are a bit more complex. They are both pointers to memory locations where the function will retrieve information (destLen starts as the available length of the destination buffer) as well as return information (dest, where the result is placed, and destLen, the amount of dest actually used).

In general, C libraries cannot deal directly with Smalltalk objects since the format is different and objects can move in memory with various garbage collection operations. As part of making the C function call, the virtual machine converts the Smalltalk objects to C data and constructs a C stack before making the C library call. For many objects this works fine; as we saw in the getcwd() example above, simple String and Integer objects are handled properly. But when an argument is a pointer to a chunk of memory in which the C library will place arbitrary data, we need to explicitly allocate that space and pass a pointer to it.

The class CByteArray represents a chunk of memory that is outside the Smalltalk object space (it is on the "heap"), and when an instance of CByteArray is passed as a #'ptr' type, the virtual machine puts a pointer to the space on the stack before making the function call. There are methods in CByteArray to place various Smalltalk objects in the allocated memory and to retrieve Smalltalk objects from the memory.

To allocate memory for the destination buffer, we can do the following:

```
dest := CByteArray gcMalloc: 100.
```

The gcMalloc constructor says to create space on the heap (outside of Smalltalk's object memory) and create a Smalltalk object (in object memory) that references the external memory. The heap memory will be automatically freed when the Smalltalk object is garbage collected. We don't need to put anything into the memory since the compress() function will not retrieve anything from the buffer. We pick a size that is enough to hold the expected result (we made an educated guess for this example; in real use we could get a better estimate by calling compressBound() with the source length).

To allocate memory for the destination size, and put a value in the location, we can do the following:

```
dest_size := CByteArray gcMalloc: 8.
dest_size uint64At: 0 put: destination size.
```

This allocates 8 bytes in the heap and puts the integer 100 (or whatever size we have allocated for the destination buffer) in that memory location (starting at a zero-based offset of 0). When we call the function we will pass a pointer to the number, not the number itself. This is so we provide a place for the function to tell us the amount of the destination buffer actually used (reusing the memory we allocated). After we make the call we can get the size back from the memory location:

```
used := dest_size uint64At: 0.
```

Once we know the amount of the destination actually used, we can extract the zip data. Note that the zip data is generic binary data, not a string, and may include bytes with a value of 0 (so cannot be treated as a C-string). Note that we are again dealing with zero-based offsets since our underlying structures are C memory:

```
compressed := destination byteArrayFrom: 0 to: used - 1.
```

We can put this all together and pass a source string to be compressed:

Example 17.4 CCallout to invoke compress()

```
| ccallout_compress source dest dest_size result used   compressed
|
ccallout_compress := CCallout
   library: (CLibrary named: '/lib/libz.so.1.2.3.3')
   name: 'compress'
   result: #'int32'
   args: #(#'ptr' #'ptr' #'const char*' #'uint64').
source := 'The quick brown fox jumped over the lazy dog'.
dest := CByteArray gcMalloc: 100.
dest_size := CByteArray gcMalloc: 8.
dest_size uint64At: 0 put: dest size.
result := ccallout_compress callWith: (Array with: dest with:
dest_size with: source with: source size).
used := dest_size uint64At: 0.
compressed := dest byteArrayFrom: 0 to: used - 1.
```

If the result is zero (Z_OK), then the function executed successfully, and compressed will reference a ByteArray that contains the compressed data.

Creating a Smalltalk class

The CHeader object can also be used to create a new Smalltalk class and automatically generate methods to invoke the C functions.

The method `CHeader >> wrapperForLibraryAt:` can be used to create a Smalltalk class with default name and methods for each function. The default name is the library name without the 'lib', so for zlib.h, the resulting class name is simply "Z".

To create Smalltalk syntax to allow arguments to be passed to the C function in the generated interface methods, each function argument is represented with "_:". So for example for the `getcwd()` function, which as two arguments, the equivalent Smalltalk method is:

`getcwd_:` *buffer* `_:` *size*

To generate a wrapper class for the zlib library, in the most simple case you could use the following code:

Example 17.5 Create wrapper class using default

```
| header wrapperClass wrapper |
header := CHeader path: '/usr/include/zlib.h'.
wrapperClass := header wrapperForLibraryAt:
    '/lib/libz.so.1.2.3.3'.
wrapperClass initializeFunctions.
UserGlobals at: wrapperClass name put: wrapperClass.
```

After this is executed, you can use a code browser to view the class-side methods that create the CCallout instances, and the instance-side methods that call the functions.

As mentioned earlier, the header file may include many functions beyond that provided in the library -- all the functions that are defined in the referenced include files. And we can call any of these functions through this library, due to the way the C function lookup occurs.

For example, the function `getpid()` is defined to take no arguments and return a 32-bit number. This makes it very easy to call once we have defined a wrapper class:

Example 17.6 Invoke Z function getpid

```
topaz 1> run
Z new getpid
%
22753
```

We probably don't want to allow the Z class to have access to every function that is included - for example, it might be better not to have access to `sethostid()`, which changes the current machine's Internet number. It's better to be more selective about what functions to include in the wrapper. It's also desirable to have a more descriptive name for the library wrapper class.

The method `CHeader>> wrapperNamed:forLibraryAt:select:` allows you to specify the name and a select block to determine the specific libraries to include. The select block should evaluate to a Boolean that indicates whether or not to include the particular function.

For example, to create a wrapper for various compress functions, you could do the following:

Example 17.7 Create wrapper class specifying name and functions

```
| header class |
UserGlobals removeKey: #'ZLib' ifAbsent: [].
header := CHeader path: '/usr/include/zlib.h'.
class := header
   wrapperNamed: 'ZLib'
   forLibraryAt: '/lib/libz.so.1.2.3.3'
   select: [:each |
      each name includesString: 'compress'].
class initializeFunctions.
UserGlobals at: class name put: class.
```

This code creates a wrapper class, ZLib, that contains only four functions: `compress()`, `uncompress()`, `compress2()`, and `compressBound()`, all the ones that happen to include the string "compress". The select block may be considerably more complex, depending on which specific libraries you want to include.

To invoke compress using the Zlib class rather than manually creating a CCallout:.

Example 17.8 Invoke Zlib function compress()

```
topaz 1> printit
| source destination dest_size result used compressed |
source := 'The quick brown fox jumped over the lazy dog'.
destination := CByteArray gcMalloc: 100.
dest_size := CByteArray gcMalloc: 8.
dest_size uint64At: 0 put: destination size.
result := ZLib new
      compress_: destination
      _: dest_size
      _: source
      _: source size.
used := dest_size int64At: 0.
compressed := destination byteArrayFrom: 0 to: used - 1.
compressed
%
x.^K.HU(,.L.VH*./.SH..P.*.-HMQ./K-R(^A..$VU*...^C.k.^P0
```

Chapter

18 External Sessions

GemStone/S 64 Bit incorporates a number of classes that facilitate spawning and managing external sessions. External sessions allow you to execute Smalltalk code in separate Gems, which may run on different servers and log in as different users to different repositories. This allows you do to things such as partitioning work among multiple gems or managing separate repositories.

Operations to create and communicate with the external sessions use the Foreign Function Interface (FFI) to access the GCI libraries, except on AIX, which uses GciInterface primitives. GciLibrary provides interface methods for all the GemBuilder for C functions.

Specifying NRS with GsNetworkResourceString
describes how to programmatically define NRS Strings

Using ExternalSessions
How to create and use external sessions

18.1 Specifying NRS with GsNetworkResourceString

GemStone uses a Network resource string, or NRS, to specify the details for the gem and stone on login. NRS strings are also used for other purposes and include a number of features; the NRS syntax is documented in the System Administration Guide, appendix C.

While you may compose strings in NRS syntax for your external session logins, the new class GsNetworkResourceString provides a way to compose NRS strings from the significant elements.

This class includes parameters that are meaningful for both Stone and Gem NRS strings, which may have a different meaning in the gem vs. in the stone, and ones which apply to one or the other,

Gem NRS methods

These methods return the NRS for a gem, using defaults as needed:

`gemNRS`
`gemNRSForNetLDI:` *nameOrPort*
`gemNRSForNetLDI:` *nameOrPort* `onHost:` *gemhostname*
`gemNRSForNetLDI:` *nameOrPort* `onHost:` *gemhostname* `gemService:` *customGemService*

nameOrPort specifies the netldi name, or the port of the netldi on the stone's host. If not provided, gs64dli is used.

gemhostname specified the name of the host on which the gem will run. If a variant without this argument is used, it will default to the stone's host.

customGemService is the name of the gem service (script). This is normally gemnetobject, but may be gemnetdebug or a customized script.

Stone NRS methods

These methods return the NRS for a gem, using defaults as needed:

`stoneNRS`
`stoneNRSForStoneName:` *aStoneName*
`stoneNRSForStoneName:` *aStoneName* `onHost:` *stoneHostName*

aStoneName is the name of the stone that you will log into. If a variant without this argument is used, it defaults to gs64stone.

stonehostname specified the name of the host on which the stone is running. If a variant without this argument is used, it will default to localhost.

GsNetworkResourceString direct protocol

The following instance methods set NRS parameters directly. You may either create an instance of GsNetworkResourceString using the above methods, and send these messages to further specify the NRS; or you may create a new instance of GsNetworkResourceString and construct it using these and other methods.

`log:`
Set the name of the log file for the Gem service. Optional; applies for the Gem's NRS.

`temporaryObjectCacheSize:`
Specify the size of the temporary object cache of the new gem. Optional; applies for the Gem's NRS.

`dir:`
Applies when starting a gem session. Set the default directory for the Gem. Optional; applies for the Gem's NRS.

`authorization:`
Sets the host UNIX user name and password. For example, 'user@password'. Applies for the Gem's NRS, if required by the NetLDI mode.

`netldi:`
Set the name or port of the netldi that will be used to service the request. Applies for the Gem's NRS, if not using the default netldi name.

node:
For a stone, sets the node that the stone is running on; when starting a gem session, the node that the gem process will be run on.

body:
For a stone, the name of the stone. For the gem, the name of the gem service. Gem services may be gemnetobject or gemnetdebug, or a custom gem service.

For example, with the Stone on a machine named santiam, to run with a Gem on the Stone's node , the Stone and Gem NRS could be defined as follows:

```
myStoneNRS := GsNetworkResourceString
   stoneNRSForStoneName: 'gs64stone'
   onHost: 'santiam.gemtalksystems.com'.

myGemNRS := GsNetworkResourceString
   gemNRSForNetldi: 'gs64ldi'
   onHost: 'santiam.gemtalksystems.com'
```

18.2 Using ExternalSessions

To use an external session, you must create an instance of GsExternalSession, set the appropriate login parameters, and login, which creates the external gem session.

After you have executed operations on the external gem, you must logout, to ensure the external gem is terminated and does not continue to use resources.

You cannot persist instances of GsExternalSession in the repository.

Setup the External Session

The login parameters you configure are the same as when logging in via topaz or other interfaces: the Stone's Network Resource String (NRS), the Gem's NRS, and the userId and password that the external session will login as. If your login requires host username and host password, these are also provided as part of the NRS arguments.

The Stone and Gem NRS may be provided as instances of the new class GsNetworkResourceString, or as strings using GemStone's standard NRS syntax.

Creating the External Session

To create the external session, create an instance and specify instances of GsNetworkResourceString or NRS strings. The following examples show equivalents using GsNetworkResourceString and NRS strings.

With GsNetworkResourceString

Note that this uses the instances of GsNetworkResourceString defined above.

```
myGsExternalSession := GsExternalSession new.
myGsExternalSession
   stoneNRS: myStoneNRS;
   gemNRS: myGemNRS;
   username: 'DataCurator';
   password: 'swordfish'.
```

Using NRS strings

```
myGsExternalSession := GsExternalSession new.
myGsExternalSession
   stoneNRS: '!@santiam.gemtalksystems.com!gs64stone';
   gemNRS: '!@santiam.gemtalksystems.com#netldi:gs64ldi!gemnetobject';
   username: 'DataCurator';
   password: 'swordfish'.
```

Log in the External Session

To login, send #login to the configured GsExternalSession:

```
myGsExternalSession login.
```

Login creates a gem session that is logged in and in transaction in the specified stone, either the same stone as the calling session or a different stone. If the external gem is logged into a stone that is in active use, you must manage the gem appropriately to avoid creating a commit record backlog in that stone; avoid leaving external gems logged in and idle, and ensure that the code you execute commit or aborts regularly.

To logout, send #logout to the logged-in GsExternalSession:

```
myGsExternalSession logout.
```

Executing Code

Code to be executed by the external session can be passed as strings or blocks. These can be executed synchronously or asynchronously.

Code in Strings

To synchronously execute code contained in a string, use the method `executeString:`.

For example:

```
myGsExternalSession executeString:
'SystemRepository fullBackupTo: ''/backups/gs/bkup14-03-23.dat'''.
```

Code in Blocks

What is actually sent to the remote session is always in the form of a String, but methods are provided that accept blocks containing the code to execute in the remote session. The source strings for these block will be passed to the remote session. This allows Smalltalk tools to manage the source, detect senders, and so on, which is not possible with strings.

To use blocks, the blocks must be able to compile in both the calling session and the remote session in which you intend them to execute, although the block's code is not necessarily meaningful in the calling session. Any variable resolution, etc. in the blocks will be resolved again in the environment of the remote session when the block is compiled after being transmitted as a string, and if the variables cannot be resolved in the remote session, it will result in an error.

Code in block can also be executed synchronously or asynchronously.

To synchronously execute code contained in a block, use:

`executeBlock:` *aNoArgBlock*
`executeBlock:` *aOneArgBlock* `with:` *aValue*
`executeBlock:` *aTwoArgBlock* `with:` *aValue* `with:` *anotherValue*
`executeBlock:` *aBlock* `withArguments:` *aCollectionOfValues*

These methods execute the source code contained in the given block, and return the result of executing that code.

When passing arguments to the block, the arguments values must be objects for which the printString allows the correct object state to be recreated in the remote session. This is true for all objects, including specials, strings, integers and floats; use caution to avoid unexpected conversion or loss of information as well as errors.

Return Values

After code is executed in the remote session, the result is returned to the calling session.

If the result of the expression is a special (Character, Boolean, SmallInteger, SmallFloat, etc.), or a String, Symbol, or ByteArray, the results are converted into the appropriate object in the calling Gem.

When result is not a special, then the OOP of the result is places in the ExportSet of the remote session. See the cautions on page 310.

Expressions that return another type of object will return an Array containing the OOP of the result. This should be avoided, except when performing additional remote operations on returned OOPs. The returned OOP is for the value of the result in the remote session, which may not exist or be resolvable in the calling session; and OOP lookup has an inherent risk of unexpected results.

Since the evaluation is done in a separate Gem process, any transient changes in the remote Gem are not visible in the calling Gem. In order for persistent changes in the remote Gem to be visible to the calling Gem, the remote Gem must commit the changes, and the calling Gem must abort.

Asynchronous Execution

The **`executeString:`** and **`executeBlock:`** methods block the calling session until execution completes. To execute the remote code asynchronously and return control immediately to the calling session, the following equivalent methods are available:

`forkString:` *aString*
`forkBlock:` *aNoArgBlock*
`forkBlock:` *aOneArgBlock* `with:` *aValue*
`forkBlock:` *aTwoArgBlock* `with:` *aValue* `with:` *anotherValue*

When you execute asynchronously, an external call is in progress, and the methods you can invoke on the remote session are limited:

`isResultAvailable`
Check whether the current call in progress has finished and save the result if it has.

`lastResult`
Answer the result received when the last `isResultAvailable` answered true, which includes after a `waitForResult` operation completed.

`waitForResult`
Wait for the external Gem to complete the current operation.

`waitForResultForSeconds:` *numSeconds*
Wait up to *numSeconds* seconds for the external Gem to complete the current operation.

`waitForResultForSeconds:` *numSeconds* `otherwise:` *aBlock*
Wait up to *numSeconds* seconds for the external Gem to complete the current operation. If the operation does not complete within that time, answer the result of evaluating aBlock.

Operations on remote objects

If you perform a remote operation that returns an OOP, you can send specific selectors to that remote object by OOP.

`send:` *anOop* `to:` *selector*
Perform the given selector on the object in the external session with the OOP *anOop*.

`send:` *selector* `to:` *anOop* `withArguments:` *anArrayOfValues*
Perform the given selector on the object represented by the given OOP, which is an OOP in the external session, and pass the Array of arguments.

The OOPs of the arguments are passed to the remote session. These arguments must be specials, or persistent objects that exist on both the calling and remote sessions, otherwise it will result in an error.

Managing Remote Sessions

Managing transaction state

Management of transaction state in the remote gem can generally be done by executing code on the remote gem. The following methods are provided for convenience.

```
GsExternalSession >> abort
GsExternalSession >> commit
```

Logging

Login and logout will output messages to stdout for the session that created the GsExternalSession; either the RPC Gem log, or the linked topaz session. You may control the location of this logging, or suppress these messages using `GsExternalSession >> suppressLogging`. However, the regular GCI login message is sent by the GCI layer, and is not affected by image-level logging control.

Breaking remote execution

You can break execution on the remote session using

```
GsExternalSession >> softbreak
GsExternalSession >> hardbreak
```

Important caution on Export Set of remote session

For objects other than specials (Integers, Characters, etc.) that are returned by the remote Gem, the remote Gem adds these objects to its export set. This includes Strings and other byte collections, Exceptions returned by the external session, and other objects that are

returned as OOPs. These OOPs remain in the export set of the remote gem, and will not be garbage collected, until that gem is logged out. These OOPS can be removed manually from the export set using Hidden Set protocol.

Although Strings and similar byte-format results and exceptions are converted into new String (or appropriate) instances in the calling Gem (with a new OOP), the OOP of the original String on the remote Gem remains in the external Gem's export set.

Exceptions

The class GciError and GciLegacyError are provided to represent errors during remote execution. If the code being executed on the remote session encounters an exception, this is raised as a GciError in the calling session, or a GciLegacyError if the with GsLegacyExternalSession.

Since remote debugging is not possible with this interface, the stack of the error is included with the error description.

For example, given the following code which triggers an error on the remote session:

```
result := [myGsExternalSession executeString: '1/0']
   on: GciError
   do: [:ex | ex description].
```

The result in the calling session will be:

```
GciError: a ZeroDivide occurred (error 2026), reason:numErrIntDi-
visionByZero, attempt to divide 1 by zero
1 AbstractException >> signal @1 line 1
    receiver a ZeroDivide occurred (error 2026), reason:numErrInt-
DivisionByZero, attempt to divide 1 by zero
2 Number >> _errorDivideByZero @5 line 6
    receiver 1
3 SmallInteger >> / @5 line 7
    receiver 1
    aNumber 0
4 Executed Code  @1 line 1
    receiver nil
5 GsNMethod class >> _gsReturnToC @1 line 1
    receiver nil
```

Chapter

19 The SUnit Framework

SUnit is a minimal yet powerful framework that supports the creation of automated unit tests. This chapter discusses the importance of repeatable unit tests and illustrates the ease of writing them using SUnit.[1]

Why SUnit?
introduces the SUnit framework and its benefit to the application developer.

Testing and Tests
describes the general goals of automated testing.

SUnit by Example
presents a step-by-step example that illustrates the use of SUnit.

The SUnit Framework
describes the core classes of the SUnit framework.

Understanding the SUnit Implementation
explores key aspects of the implementation by following the execution of a test and test suite.

19.1 Why SUnit?

Writing tests is an important way of investing in the future reliability and maintainability of your code. Tests should be repeatable, automated, and cover a precise functionality to maximize their potential.

SUnit was developed originally by Kent Beck and was extended by Joseph Pelrine and others. The interest in SUnit is not limited to the Smalltalk community. Indeed, legions of developers understand the power of unit testing and versions of XUnit (as the general framework is called) exist in many other languages.

1. This chapter is adapted from "SUnit Explained" by Stéphane Ducasse (http://www.iam.unibe.ch/~ducasse/Programmez/OnTheWeb/Eng-Art8-SUnit-V1.pdf) and is used by permission.

Testing and building regression test suites is not new; it is common knowledge that regression tests are a good way to catch errors. Extreme Programming has brought a new emphasis to this somewhat neglected discipline by making testing a foundation of its methodology. The Smalltalk community has a long tradition of testing, due to the incremental development supported by its programming environment. However, once you write tests in a workspace or as example methods, there is no easy way to keep track of them and to automatically run them. Unfortunately, tests that you cannot automatically run are less likely to be run. Moreover, having a code snippet to run in isolation often does not readily indicate the expected result. That's why SUnit is interesting—it provides a code framework to describe the context of your tests and to run them automatically. In less than two minutes, you can write tests using SUnit that become part of an automated test suite. This represents a vast improvement over writing small code snippets in an ephemeral workspace.

19.2 Testing and Tests

Many traditional development methodologies include testing as a step that follows coding, and this step is often cut short when time pressures arise. Yet development of automated tests can save time, since having a suite of tests is extremely useful and allows one to make application changes with much higher confidence.

Automated tests play several roles. First, they are an active and *always synchronized* documentation of the functionality they cover. Second, they represent the developer's confidence in a piece of code. Tests help you quickly find defects introduced by changes to your code. Finally, writing tests at the same time or even before writing code forces you to think about the functionality you want to design. By writing tests first, you have to clearly state the context in which your functionality will run, the way it will interact with other code, and, more important, the expected results. Moreover, when you are writing tests, you are your first client and your code will naturally improve.

The culture of tests has always been present in the Smalltalk community; a typical practice is to compile a method and then, from a workspace, write a small expression to test it. This practice supports the extremely tight incremental development cycle promoted by Smalltalk. However, because workspace expressions are not as persistent as the tested code and cannot be run automatically, this approach does not yield the maximum benefit from testing. Moreover, the context of the test is left unspecified so the reader has to interpret the obtained result and assess whether it is right or wrong.

It is clear that we cannot tests all the aspects of an application. Covering a complete application is simply impossible and should not be goal of testing. Even with a good test suite, some defect can creep into the application and be left hidden waiting for an opportunity to damage your system. While there are a variety of test practices that can address these issues, the goal of *regression* tests is to ensure that a previously discovered and fixed defect is not reintroduced into a later release of the product.

Writing good tests is a technique that can be easily learned by practice. Let us look at the properties that tests should have to get a maximum benefit:

- Repeatable. We should be able to easily repeat a test and get the same result each time.
- Automated. Tests should be run without human intervention. You should be able to run them during the night.

- Tell a story. A test should cover one aspect of a piece of code. A test should act as a specification for a unit of code.
- Resilient. Changing the internal implementation of a module should not break a test. One way to achieve this property is to write tests based on the interfaces of the tested functionality.

In addition, for test suites, the number of tests should be somehow proportional to the bulk of the tested functionality. For example, changing one aspect of the system might break *some* tests, but it should not break *all* the tests. This is important because having 100 tests broken should be a much more important message for you than having 10 tests failing.

By using "test-first" or "test-driven" development, eXtreme Programming proposes to write tests even before writing code. While this is counter-intuitive to the traditional "design-code-test" mindset, it can have a powerful impact on the overall result. Test-driven development can improve the design by helping you to discover the needed interface for a class and by clarifying when you are done (the tests pass!).

The next section provides an example of an SUnit test.

19.3 SUnit by Example

Before going into the details of SUnit, let's look at a step-by-step example. The example in this section tests the class Set, and is included in the SUnit distribution so that you can read the code directly in the image.

Step 1: Define the Class ExampleSetTest

Example 19.1 defines the class ExampleSetTest, a subclass of TestCase.

Example 19.1

```
TestCase subclass: 'ExampleSetTest'
   instVarNames: #( full empty)
   classVars: #()
   classInstVars: #()
   poolDictionaries: #()
   inDictionary: Globals
```

The class ExampleSetTest groups all tests related to the class Set. It establishes the context of all the tests that we will specify. Here the context is described by specifying two instance variables, `full` and `empty`, that represent a full and empty set, respectively.

Step 2: Define the Method setUp

Example 19.2 presents the method `setUp`, which acts as a context definer method or as an initialize method. It is invoked before the execution of any test method defined in this class. Here we initialize the `empty` instance variable to refer to an empty set, and the `full` instance variable to refer to a set containing two elements.

Example 19.2

```
ExampleSetTest>>setUp
   empty := Set new.
   full := Set with: 5 with: #abc.
```

This method defines the context of any tests defined in the class. In testing jargon, it is called the *fixture* of the test.

Step 3: Define Three Test Methods

Example 19.3 defines three methods on the class ExampleSetTest. Each method represents one test. If your test method names begin with `test`, as shown here, the framework will collect them automatically for you into test suites ready to be executed.

Example 19.3

```
ExampleSetTest>>testIncludes
   self assert: (full includes: 5).
   self assert: (full includes: #abc).

ExampleSetTest>>testOccurrences
   self assert: (empty occurrencesOf: 0) = 0.
   self assert: (full occurrencesOf: 5) = 1.
   full add: 5.
   self assert: (full occurrencesOf: 5) = 1.

ExampleSetTest>>testRemove
   full remove: 5.
   self assert: (full includes: #abc).
   self deny: (full includes: 5).
```

The `testIncludes` method tests the `includes:` method of a Set. After running the `setUp` method in Example 19.2, sending the message `includes: 5` to a set containing 5 should return `true`.

Next, `testOccurrences` verifies that there is exactly one occurrence of 5 in the `full` set, even if we add another element 5 to the set.

Finally, `testRemove` verifies that if we remove the element 5 from a set, that element is no longer present in the set.

Step 4: Execute the Tests

Now we can execute the tests, using either Topaz or one of the GemBuilder interfaces. To run your tests, execute the following code:

```
(ExampleSetTest selector: #testRemove) run.
```

Alternatively, you can execute this expression:

```
ExampleSetTest run: #testRemove.
```

Developers often include such an expression as a comment, to be able to run them while browsing. See Example 19.4.

Example 19.4

```
ExampleSetTest>>testRemove
   "self run: #testRemove"
   full remove: 5.
   self assert: (full includes: #abc).
   self deny: (full includes: 5).
```

To debug a test, use one of the following expressions:

```
(ExampleSetTest selector: #testRemove) debug.
```

or

```
ExampleSetTest debug: #testRemove.
```

Examining the Value of a Tested Expression

The method `TestCase>>assert:` requires a single argument, a boolean that represents the value of a tested expression. When the argument is true, the expression is considered to be correct, and we say that the test is valid. When the argument is false, then the test failed. The method `deny:` is the negation of `assert:`. Hence

```
aTest deny: anExpression.
```

is equal to

```
aTest assert: anExpression not.
```

Finding Out If an Exception Was Raised

SUnit recognizes two kinds of defects: not getting the correct answer (a failure) and not completing the test (an error). If it is anticipated that a test will not complete, then the test should raise an exception. To test that exceptions have been raised during the execution of an expression, SUnit offers two methods, `should:raise:` and `shouldnt:raise:`. See Example 19.5.

Example 19.5

```
ExampleSetTest>>testIllegal
   self should: [empty at: 5] raise: Error.
   self should: [empty at: 5 put: #abc] raise: Error.
```

In the example provided by SUnit, the exception is provided via the TestResult class (Example 19.6). Because SUnit runs on a variety of Smalltalk dialects, the SUnit framework factors out the variant parts (such as the name of the exception). If you plan to write tests that are intended to be cross-dialect, look at the class TestResult.

Example 19.6

```
ExampleSetTest>>testIllegal
   self should: [empty at: 5] raise: TestResult error.
   self should: [empty at: 5 put: #abc] raise: TestResult error.
```

Because GemStone Smalltalk has a legacy exception framework that uses numbers to identify exceptions, a subclass of TestCase is provided, GSTestCase, which overrides `should:raise:` to allow a number argument for the expected error type.

Example 19.7

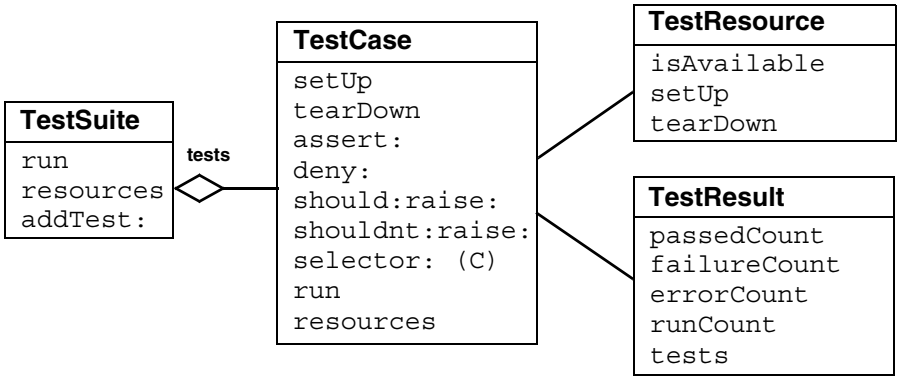
```
GSExampleSetTest>>testIllegal
   self should: [empty at: 5] raise: 2007.
   self should: [empty at: 5 put: #abc] raise: 2007.
```

Having provided an example of writing and running a test, we now turn to an investigation of the framework itself.

19.4 The SUnit Framework

SUnit is implemented by four main classes: TestSuite, TestCase, TestResult, and TestResource. See Figure 19.1. (Note that this is an object composition diagram, not a class hierarchy diagram.)

Figure 19.1 The SUnit Core Classes



TestSuite

The class TestSuite represents a collection of tests. An instance of TestSuite contains zero or more instances of subclasses of TestCase and zero or more instances of TestSuite. The classes TestSuite and TestCase form a composite pattern in which TestSuite is the composite and TestCase is the leaf.

TestCase

The class TestCase represents a family of tests that share a common context. The context is specified by instance variables on a subclass of TestCase and by the specialization method `setUp`, which initializes the context in which the test will be executed. The class TestCase also defines the method `tearDown`, which is responsible for cleanup, including releasing the objects allocated by `setUp`. The method `tearDown` is invoked after the execution of every test.

TestResult

The class TestResult represents the results of a TestSuite execution. This includes a description of which tests passed, which failed, and which had errors.

TestResource

Recall that the `setUp` method is used to create a context in which the test will run. Often that context is quite inexpensive to establish, as in Example 19.2 (on page 316), which creates two instances of Set and adds two objects to one of those instances.

At times, however, the context may be comparatively expensive to establish. In such cases, the prospect of re-establishing the context for each run of each test might discourage frequent running of the tests. To address this problem, SUnit introduces the notion of a *resource* that is shared by multiple tests.

The class TestResource represents a resource that is used by one or more tests in a suite, but instead of being set up and torn down for each test, it is established once before the first test and reset once after the last test. By default, an instance of TestSuite defines as its resources the list of resources for the TestCase instances that compose it.

As shown in Example 19.8, a resource is identified by overriding the class method `resources`. Here, we define a subclass of TestResource called MyTestResource. We associate it with MyTestCase by overriding the class method `resources` to return an array of the test classes to which it is associated.

Example 19.8

```
MyTestCase class>>resources
   "associate a resource with a testcase"
   ^ Array with: MyTestResource.
```

As with a TestCase, we use the method `setUp` to define the actions that will be run during the setup of the resource.

19.5 Understanding the SUnit Implementation

Let's now look at some key aspects of the implementation by following the execution of a test. Although this understanding is not necessary to use SUnit, it can help you to customize SUnit.

Running a Single Test

To execute a single test, we evaluate the expression

```
(TestCase selector: aSymbol) run.
```

The method `TestCase>>run` creates an instance of `TestResult` to contain the result of the executed tests, and then invokes the method `TestCase>>run:`, which in turn invokes the method `TestResult>>runCase:`. See Figure 19.2.

Figure 19.2 TestCase instance methods `run` and `run:` (source code)

```
TestCase>>run
   | result |
   result := TestResult new.
   self run: result.
      ensure: [TestResource resetResources: self resources].
   ^result.

TestCase>>run: aResult
   aResult runCase: self.
```

The `runCase:` method (Figure 19.3) invokes the method `TestCase>>runCase`, which executes a test. Without going into the details, `TestCase>>runCase` pays attention to the possible exception that may be raised during the execution of the test, invokes the execution of a TestCase by calling the method `runCase`, and counts the errors, failures, and passed tests.

Figure 19.3 TestResult instance method `runCase:` (source code)

```
TestResult>>runCase: aTestCase
        [aTestCase runCase.
        self addPass: aTestCase]
           on: self class failure , self class error
           do: [:ex | ex sunitAnnounce: aTestCase toResult: self]
```

As shown in Figure 19.4, the method `TestCase>>runCase` calls the methods `setUp` and `tearDown`.

Figure 19.4 TestCase instance method `runCase` (source code)

```
TestCase>>runCase
        self resources do: [:each | each availableFor: self].
        [self setUp.
```

```
        self performTest]
        ensure: [self tearDown]
```

Running a TestSuite

To execute more than a single test, we invoke the method `TestSuite>>run` on a TestSuite (see Figure 19.5). The class TestCase provides the functionality to build a test suite from its methods. The expression `MyTestCase suite` returns a suite containing all the tests defined in the class `MyTestCase`.

The method `TestSuite>>run` creates an instance of TestResult, verifies that all the resource are available, then invokes the method `TestSuite>>run:` to run all the tests that compose the test suite. All the resources are then reset.

Figure 19.5 TestSuite instance methods `run` and `run:` (source code)

```
TestSuite>>run
        | result |
        result := TestResult new.
        [self run: result]
           ensure: [TestResource resetResources: self resources].
        ^result

TestSuite>>run: aResult
        self tests do: [:each |
           self sunitChanged: each.
           each run: aResult]
```

The class TestResource and its subclasses use the class method `current` to keep track of their currently created instances (one per class) that can be accessed and created. This instance is cleared when the tests have finished running and the resources are reset. The resources are created as needed. See Figure 19.6.

Figure 19.6 TestResource class methods `isAvailable` and `current` (source code)

```
        TestResource class>>isAvailable
           ^self current notNil

        TestResource class>>current
           current isNil ifTrue: [current := self new].
           ^current
```

19.6 For More Information

To continue your exploration of repeatable unit testing, visit the Camp Smalltalk SUnit site (http://sunit.sourceforge.net). The SUnit site provides information about SUnit development efforts, along with downloads, documentation, and other materials of interest.

You may also find these books helpful:

Beck, Kent. *Test-Driven Development: By Example.* Addison-Wesley, 2003.

Beck, Kent, and Cynthia Andres. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2004.

Fowler, Martin, and Kent Beck. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

Chapter

A GemStone Smalltalk Syntax

This appendix outlines the syntax for GemStone Smalltalk and introduces the important kinds of GemStone Smalltalk objects.

A.1 GemStone and ANSI Smalltalk

GemStone's programming language, GemStone Smalltalk, is a dialect of the Smalltalk programming language. The Smalltalk language standard is defined by an ANSI Smalltalk standard. While GemStone follows this standard, there are places where either for historical reasons or by choice, GemStone Smalltalk does not follow the ANSI standard.

Some known places in which GemStone Smalltalk does not conform to the ANSI standard:

- Arrays sizes are not fixed; an Array may increase in size by 1 when an operation assign to an index not more than 1 larger than the current size.
- Array constructors using {} are not part of the standard.
- `DateAndTime asSeconds` does not print fractional seconds.
- `Integer >> asInteger` truncates rather than rounds.
- The Fixed point literal syntax with 'p' is not part of the standard.

A.2 GemStone Smalltalk

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

Classes are created using a number of class creation methods, defined on the class Class. For example, following message expression makes a new subclass of class Object, the class at the top of the class hierarchy:

```
Object subclass: 'Animal'
      instVarNames: #()
      classVars: #()
      classInstVars: #()
      poolDictionaries: {}
      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 1024 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. For more on class creation, see Chapter 2.

Case-Sensitivity

GemStone Smalltalk is case-sensitive; that is, names such as "SuperClass," "superclass," and "superClass" are treated as unique items by the GemStone 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

GemStone 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 page 327). Within a GemStone Smalltalk string literal, a "comment" becomes part of the string.
- When it immediately follows a dollar sign ($). GemStone Smalltalk interprets the first character after a dollar sign as a data object called a character literal (see page 327).

A comment terminates tokens such as numbers and variable names. For example, GemStone 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 GemStone Smalltalk expressions and tell you something about their behavior.

Kinds of Expressions

A GemStone Smalltalk expression can contain 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 GemStone 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 other 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
```

The literal numeric type GemStone/S 64 Bit supports are:

- “e”, “E”, “d” and “D” for floating point literals (SmallDouble or Float)
- “f” and “F” for DecimalFloat literals
- “s” for ScaledDecimal literals
- “p” for FixedPoint literals

For details, see “GemStone Smalltalk Lexical Tokens” on page 346.

To represent a number in a nondecimal base literally, write the number’s base (in decimal), followed by the radix “r” or character "#", and then the number itself. Here, for example, is how you could write octal 23 and hexadecimal FF:

```
8#23
16rFF
```

The largest radix available is 36.

Character Literals

A Smalltalk character literal represents a character, such as 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.

GemStone Smalltalk interprets this statement, for example, as a representation of ASCII character 32:

```
$ . "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, GemStone 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.'
```

When you want to include apostrophes in a literal string, double them:

```
'You can''t make omelettes without breaking eggs.'
```

GemStone Smalltalk faithfully preserves control characters when it compiles literal strings. The following example creates a String containing a line feed (ASCII 10), the end-of-line character:

```
'Control characters such as line feeds
 are significant in literal strings.'
```

Strings may hold characters with values up to 255, that is, characters that can be representing in a single byte. Characters themselves may have values much higher. If a string includes any characters larger than 255, it is converted to a DoubleByteString. If any of the characters require more than two bytes, it becomes a QuadByteString. For example, this is a DoubleByteString:

```
'Škoda'
```

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.1

```
#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 preceded by a pound sign (#) and must also be delimited by single quotation marks, as described in the "String Literals" discussion. For example:

```
#'Gone With the Wind'
```

As with strings that contain characters that require more than a byte to represent, DoubleByteSymbol and QuadByteSymbol are used for symbol literals that include characters with values over 255.

Array Literals

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, other literal Arrays, and three "special literals" (true, false, nil). 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, a Number, and a Boolean:

```
#('string one' #symbolOne $c 4 true)
```

Besides Array literals, you may also specify Array constructors in your code, which are used similarly, but follow quite different rules. For a discussion of array constructors, see page 335.

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 (sometimes called "camelcase"). 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 1024 characters.

Declaring Temporary Variables

GemStone 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 GemStone, 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.2

```
| myTempVariable |
   myTempVariable := 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 GemStone Smalltalk uses for symbol resolution. For example:

Example A.3

```
| myTempVariable |
myTempVariable := 2.
UserGlobals at: #MyPermanentVariable put: myTempVariable.
```

Subsequent references to MyPermanentVariable return the value 2.

Pseudovariables

You can change the objects to which most variable names refer simply by assigning them new objects. However, five GemStone 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**. **nil** is an instance of UndefinedObject.

true

Refers to the object representing logical truth. **true** is an instance of Boolean.

false

Refers to the object representing logical false. **false** is an instance of Boolean.

self

Refers to the receiver of the message, which differs according to the context. **self** may be used anywhere a method argument or method temporary would be used, except **self** is not allowed on the left side of an assignment. When **self** is used in code that is not part of a method, it resolves to **nil**.

super

Refers to the receiver of the message, but the search for the method to execute will start in the superclass of the class in which the sending method was compiled. **super** may only be used as the receiver of a message send, in code within a method.

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

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 GemStone 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 and Optimized Selectors

GemStone represents selectors internally as symbols, and almost all symbols that confirm to the unary, binary, or keyword selector patterns are acceptable as a selectors. For details on legal selectors, see the BNF on page 344.

There are a few selectors that have been reserved for the sole use of the GemStone kernel classes. The compiler will not allow you to compile methods with reserved selectors.

Those selectors are:

```
__inProtectedMode  _and:                _downTo:by:do:
_downTo:do:        _gsReturnNothingEnableEvents
_gsReturnNoResult  _isArray             _isExceptionClass
_isExecBlock       _isFloat             _isInteger
_isNumber          _isOneByteString     _isRange
_isRegexp          _isRubyHash          _isScaledDecimal
_isSmallInteger    _isSymbol            _leaveProtectedMode
_or:               _stringCharSize      ~~
and:               ==                   ifFalse:
ifFalse:ifTrue:    ifNil:               ifNil:ifNotNil:
ifNotNil:          ifNotNil:ifNil:      ifTrue:
ifTrue:ifFalse:    isKindOf:            or:
timesRepeat:       to:by:do:            to:do:
untilFalse         untilFalse:          untilTrue
untilTrue:         whileFalse           whileFalse:
whileTrue          whileTrue:           repeat
```

In addition, the following methods are optimized in the class SmallInteger:

```
+        -      *      >=     =
```

You can redefine the optimized methods above in your application classes, but redefinitions in the class SmallInteger are ignored.

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.4

```
| arrayOfStrings |
arrayOfStrings := Array new: 4.
arrayOfStrings at: (2 + 1) put: 'Curly'.
"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 GemStone 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; GemStone 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, GemStone 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. GemStone 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.5

```
| arrayOfPoets |
arrayOfPoets := Array new.
(arrayOfPoets add: 'cummings'; add: 'Byron'; add: 'Rimbaud';
yourself)
```

is a cascaded message expression that is equivalent to this series of statements:

Example A.6

```
| 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

Most of the syntax described in this chapter so far is standard Smalltalk syntax. However, GemStone Smalltalk also includes a syntactic construct called a *Array constructor*. An Array constructor is similar to a literal array, but its elements can be written as nonliteral expressions as well as literals. GemStone 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 braces and have their elements delimited by periods.

Example A.7 shows an Array constructor whose last element, represented by a message expression, has the value 4.

Example A.7

```
"An Array constructor"
{'string one' . #SymbolOne . $c . 2+2}
```

NOTE

The Array constructor is not part of the Smalltalk standard. You should avoid its use in any code that might be ported to an other Smalltalk dialect. Instead, use a message send constructor such as `Array class >> #with:`*. See Example A.8.*

Example A.8

```
Array with: 'string one' with: #symbolOne with: $c with: 2+2
```

Because any valid GemStone 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.9

```
| 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 in Example A.10.

Example A.10

```
| huh |
huh := 88.
#( 123 huh 456 )

[20176897 sz:3 cls: 66817 Array] an Array
  #1 [986 sz:0 cls: 74241 SmallInteger] 123 == 0x7b
  #2 [27086593 sz:3 cls: 110849 Symbol] huh
  #3 [3650 sz:0 cls: 74241 SmallInteger] 456 == 0x1c8
```

The same declaration used in an array constructor, however, produces an array of three SmallIntegers:

Example A.11

```
| huh |
huh := 88.
{ 123 . huh . 456 }

[20192001 sz:3 cls: 66817 Array] an Array
  #1 [986 sz:0 cls: 74241 SmallInteger] 123 == 0x7b
  #2 [706 sz:0 cls: 74241 SmallInteger] 88 == 0x58
  #3 [3650 sz:0 cls: 74241 SmallInteger] 456 == 0x1c8
```

Path Expressions

With the exception of Array constructors, most of the syntax described in this chapter so far is standard Smalltalk syntax. GemStone 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 the value of `last`:

```
anEmployee.name.last
```

NOTE

Use paths only for their intended purposes. Although you can use a path anywhere an expression is acceptable in a GemStone 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 GemStone 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.12

```
| 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.13

```
"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.

A.3 Blocks

A GemStone 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 GemStone Smalltalk, especially in building control structures.

A literal block is delimited by brackets and contains one or more GemStone 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 expression.

```
[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.14

```
| 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.15

```
| 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 can 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]
```

To execute a block that takes an argument, send it the keyword message `value:` *anArgument*. For example:

Example A.16

```
| 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.17

```
| 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 GemStone Smalltalk discards it after the block finishes execution. Example A.18 illustrates this:

Example A.18

```
| 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.19

```
"The following expression attempts an invalid assignment
 to a block variable."
[:blockVar | blockVar := blockVar * 2] value: 10
```

Blocks and Conditional Execution

Most computer languages, GemStone 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.

GemStone Smalltalk flow of control statements rely on blocks because blocks so conveniently encapsulate sequences of instructions. GemStone 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. GemStone Smalltalk also provides a control structure that executes a block a specified number of times.

Conditional Selection

You will often want GemStone Smalltalk to execute a block of code only if some condition is true or only if it is false. GemStone Smalltalk provides the messages `ifTrue:` *aBlock* and `ifFalse:` *aBlock* for that purpose. Example A.20 contains both of these messages:

Example A.20

```
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, GemStone Smalltalk initially evaluates the expression `(5 = 5)`. That expression returns the value true (a Boolean), to which GemStone Smalltalk then sends the selector `ifTrue:`. The receiver (true) looks at itself to verify that it is, indeed, the object true. Because it is, it proceeds to execute the block passed as an argument to `ifTrue:`, 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 Example A.20, the expressions `(5 = 5)` and `(5 > 10)` returned true and false, respectively, because GemStone Smalltalk numbers know how to compute and return those values when they receive messages such as `=` and `>`.

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.21

```
2 < 5 ifTrue: ['two is less than five'].
two is less than five
2 < 5 ifFalse: ['two is not less than five'].
nil
```

However, GemStone 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 GemStone Smalltalk executes depends upon whether the receiver is true or false. In Example A.22, the receiver is true:

Example A.22

```
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 1 to a variable until the variable equals 5:

Example A.23

```
| 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.24

```
| totalPayroll numEmployees salariesAdded standardSalary |
totalPayroll := 0.00.
salariesAdded := 0.
numEmployees := 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 < numEmployees] whileTrue:
      [totalPayroll := totalPayroll + standardSalary.
      salariesAdded := salariesAdded + 1].
totalPayroll
2.0000000000000000E05
```

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 is equivalent to Example A.23, but uses `untilTrue` (rather than `whileFalse:`).

Example A.25

```
| sum |

sum := 0.
[sum := sum + 1. sum = 5] untilTrue.
sum

5
```

When GemStone 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 5; since it isn't, that statement returns false, and GemStone Smalltalk executes the block again. GemStone Smalltalk continues to reevaluate the block as long as the last statement returns false (that is, while `sum` is not equal to 5).

The descriptions of classes Boolean and Block in the image describe these flow of control messages and others.

Formatting Code

GemStone Smalltalk is a free-format language. A space, tab, line feed, form feed, or carriage return affects the meaning of a GemStone 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.26

```
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.4 GemStone 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 zero or one 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 GemStone Smalltalk syntactic productions in Figure A.1, white space is allowed between tokens. White space is required before and after the '_' character.

Figure A.1 GemStone Smalltalk BNF

```
ArrayBuilder = '#[' [ AExpression { ',' AExpression } ] ']'
      (exists only if System configurationAt:#GemConvertArrayBuilder is true)
ByteArrayLiteral = '#' '[' [ Number { Number } ] ']'
      (exists only if System configurationAt:#GemConvertArrayBuilder is false)
Assignment = VariableName ':=' Statement
AExpression = Primary [ AMessage { ';' ACascadeMessage } ]
ABinaryMessage = [ EnvSpecifier | RubyEnvSpecifier ] ABinarySelector Primary
      [ UnaryMessages ]
ABinaryMessages = ABinaryMessage { ABinaryMessage }
ACascadeMessage = UnaryMessage | ABinaryMessage | AKeyWordMessage
AKeyWordMessage = [ EnvSpecifier | RubyEnvSpecifier  ] AKeyWordPart { AKeyWordPart }
AKeyWordPart = KeyWord Primary UnaryMessages { ABinaryMessage }
AMessage = [UnaryMessages] [ABinaryMessages] [AKeywordMessage]
Array = '(' { ArrayItem } ')'
ArrayLiteral = '#' Array
CurlyArrayBuilder = '{' [ AExpression { '.' AExpression } ] '}'
ArrayItem = Number | Symbol | SymbolLiteral | StringLiteral |
      CharacterLiteral | Array | ArrayLiteral
BinaryMessage = [ EnvSpecifier | RubyEnvSpecifier ] BinarySelector Primary
      [ UnaryMessages ]
BinaryMessages = BinaryMessage { BinaryMessage }
BinaryPattern = BinarySelector VariableName
Block = '[' [ BlockParameters ] [ Temporaries ] Statements ']'
BlockParameters = { Parameter } '|'
CascadeMessage = UnaryMessage | BinaryMessage | KeyWordMessage
Expression = Primary [ Message { ';' CascadeMessage } ]
KeyWordMessage = [ EnvSpecifier | RubyEnvSpecifier ] 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 = [ Pragmas ] [ Temporaries ] [ Statements ]
NegNumber = '-' Number
Operand = Path | Literal | Identifier
Operator = '=' | '==' | '<' | '>' | '<=' | '>=' | '~=' | '~~'
ParenStatement = '(' Statement ')'
Predicate = ( AnyTerm | ParenTerm ) { '&' Term }
Primary = ArrayBuilder | CurlyArrayBuilder | Literal | Path | Block | SelectionBlock |
      ParenStatement | VariableName
Primitive = '<' [ 'protected' | 'unprotected' ] [ 'primitive:' Digits ] '>'
Pragmas =   Pragma [ Pragma ]
Pragma = '< PragmaBody '>'
PragmaBody =  UnaryPragma | KeywordPragma
UnaryPragma  = SpecialLiteral | UnaryPragmaIdentifier
KeywordPragma = PragmaPair [ PragmaPair ]
PragmaPair =  [ KeywordNotPrimitive | BinarySelector ] PragmaLiteral
      KeywordNotPrimitive  is any  Keyword  other than  'primitive:'
UnaryPragmaIdentifier is any  Identifier  except 'protected', 'unprotected',
      'requiresVc'
PragmaLiteral = Number | NegNumber | StringLiteral | CharacterLiteral |
      SymbolLiteral | SpecialLiteral
SelectionBlock = '{' Parameter } '|' Predicate '}'
Statement = Assignment | Expression
Statements = { [ Pragmas] { Statement '.' } } [ Pragmas] [ ['^'] Statement ['.' [
      Pragmas] ]]
Temporaries = '|' { VariableName } '|'
```

```
ParenTerm = '(' AnyTerm ')'
Term = ParenTerm | Operand
AnyTerm = Operand [ Operator Operand ]
UnaryMessage = [ EnvSpecifier | RubyEnvSpecifier ] Identifier
UnaryMessages = { UnaryMessage }
UnaryPattern = Identifier
```

GemStone Smalltalk lexical tokens are shown in Figure A.2. No white space is allowed within lexical tokens.

Figure A.2 GemStone Smalltalk Lexical Tokens

```
ABinarySelector = any BinarySelector except comma
BinaryExponent = ( 'e' | 'E' | 'd' | 'D' ) ['-'] Digits
BinarySelector =  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}
EndOfSource = the end of the method source string
Exponent = BinaryExponent | DecimalExponent | ScaledDecimalExponent |
       FixedPointExponent
FractionalPart = '.' Digits [Exponent]
FixedPointExponent =  'p' [ ['-'] Digits ]
Identifier =  SingleLetterIdentifier | MultiLetterIdentifier
KeyWord = Identifier ':'
Letter = 'A' | 'B' | ... | 'Z' | 'a' | 'b' | ... | 'z' | '_'
MultiLetterIdentifier = Letter { Letter | Digit }
Number = RadixedLiteral | NumericLiteral
Numeric = Digit | 'A' | 'B' | ... | 'Z'
NumericLiteral = Digits ( [FractionalPart] | [Exponent] )
Numerics = Numeric { Numeric }
Parameter = ':' VariableName
       (white space allowed between : and variableName)
Path = Identifier '.' PathIdentifier { '.' PathIdentifier }
PathIdentifier  =  Identifier |  '*'
EnvSpecifier = '@env' Digits ':'
       (no white space before or after Digits)
RubyEnvSpecifier '@ruby' Digits ':'
       (other keyword tokens allowed after RubyEnvSpecifier)
RadixedLiteral = Digits ( '#' | 'r' ) ['-'] Numerics
ScaledDecimalExponent =  's' [ ['-'] Digits ]
ScdExponTerminator = '"' | WhiteSpace | ',' | ')' | ']' | '}' | '.' | ';' |
       EndOfSource
SelectorCharacter = '+' | '-' | '\' | '*' | '~' | '<' | '>' | '='
       | '|' | '/' | '&' | '@' | '%' | ',' | '?' | '!'
SingleLetter 'A' | 'B' | ... | 'Z' | 'a' | 'b' | ... | 'z'
SingleLetterIdentifier =  SingleLetter
SpecialLiteral = 'true' | 'false' | 'nil' | '_remoteNil'
StringLiteral = "'" { Character | "''" } "'"
Symbol = Identifier | BinarySelector | ( Keyword { Keyword } )
SymbolLiteral = '#' ( Symbol | StringLiteral )
VariableName = Identifier
```

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z