GemStone®

GemStone/S 64 Bit Programming Guide

Version 3.0

June 2011

vmware®

GEMSTONE S™ 64

INTELLECTUAL PROPERTY OWNERSHIP

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, Inc.
15220 NW Greenbrier Parkway
Suite 150
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; the GemStone Smalltalk programming language; and may also be used to refer to the company, previously GemStone Systems, Inc., now a division of VMware, Inc.

Typographical Conventions

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

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

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

```
numericArray at: 1
  12486
```

Executing the Examples

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

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

- If you are using GemBuilder for Smalltalk, you can use the editing and execution facilities provided by a GemStone Browser or workspace. A browser makes it easier to define classes and methods by presenting templates for these operations. Once you've filled out the templates, a browser internally builds and executes GemStone Smalltalk expressions to compile the classes and methods. A workspace makes it easier to compile and execute fragments of GemStone Smalltalk code interactively, and see the results immediately using the *GS-print it* command.
- You can also enter your GemStone Smalltalk method code through the Topaz programming environment. Topaz requires a few extra commands to begin and end an example. To identify code as constituting a method, for instance, you'll add a couple of simple non-GemStone Smalltalk directives such as "**`method:`**." These tell Topaz to treat the indicated text as a method to be compiled and installed in a class.

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

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

Other 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.

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

GemStone Website

http://support.gemstone.com

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

- **Documentation** for released versions of all GemStone products, in PDF form.
- **Downloads** and **Patches**, including past and current versions of GemBuilder for Smalltalk.
- **Bugnotes**, identifying performance issues or error conditions you should be aware of.
- **TechTips**, providing information and instructions that are not otherwise included in the documentation.
- **Compatibility matrices**, listing supported platforms for GemStone 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 support may be submitted online or by telephone. We recommend you use telephone contact only for serious requests that require immediate attention, such as a production system down. The support website is the preferred way to contact Technical Support.

Website: http://techsupport.gemstone.com

Email: techsupport@gemstone.com

Telephone: (800) 243-4772 or (503) 533-3503

When submitting a request, please include the following information:

- Your name, company name, and GemStone server license number.
- The versions of all related GemStone 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 VMware/GemStone holidays.

24x7 Emergency Technical Support

GemStone Technical Support 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, if they encounter problems that cause their production application to go down, or that have the potential to bring their production application down. For more details, contact your GemStone account manager.

Training and Consulting

Consulting is available to help you succeed with GemStone products. Training for GemStone software is available at your location, and training courses are offered periodically at our offices in Beaverton, Oregon. Contact your GemStone account representative for more details or to obtain consulting services.

Contents

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

1.1 Overview of the GemStone System

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

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

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

1.2 Multi-User Object Server

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, and shared memory and SMP can be leveraged.

Multiple user sessions can be active at the same time, and each user may have multiple sessions open. A flexible naming scheme allows separate or shared namespaces for individual users. Coherent groups of objects can be distributed through replication. Changes 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.

1.3 Programmable Server Object System

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

This manual discusses the use of GemStone Smalltalk for system and application development, particularly those aspects of GemStone Smalltalk that are unique to running in a multi-user, secure, transactional system. See the *System Administration Guide for GemStone/S 64 Bit* for more information about system administration functions.

1.4 Partitioning of Applications Between Client and Server

GemStone applications can access objects and run their methods from a number of languages, including Smalltalk, C, 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, known collectively as GemBuilder, allow you to move objects between an application program and the GemStone repository, and to connect client objects to GemStone objects. GemBuilder also provides remote messaging capabilities, client replicates, and synchronization of changes.

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

GemBuilder for C is a library of C functions that provide a bridge between an application's C code and the GemStone object repository. You can work with GemStone objects by importing them into the C program using structural access or

by sending messages to objects in the repository through GemStone Smalltalk. You can also call C routines from within GemStone Smalltalk methods.

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

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

1.5 Large-Scale Repository

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

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

1.6 Queries and Indexes

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

The power and flexibility of GemStone Smalltalk allow you to perform regular and associative access queries against very large collections. Because you can represent information in forms that mirror the information's natural structure, the translation of user requests into executable queries can be much easier in GemStone. You do not need to translate users' keystrokes or menu selections into relational algebra formulas, calculus expressions and procedural statements before the query can be executed. See Chapter 6, "Querying."

1.7 Transactions and Concurrency Control

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

GemStone provides two approaches to managing concurrent transactions:

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

competition for shared information in your application, or your application can't tolerate even an occasional inability to commit, using locks may be your best choice.

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

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

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

See Chapter 7, "Transactions and Concurrency Control."

1.8 Connections 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 are written in C and linked to GemStone modules. With userActions, you can access or generate external information and bring it into GemStone as objects, which can then be committed and made available to other users. GemBuilder for C is used to write userActions in C and add them to GemStone Smalltalk, according to rules described in the *GemBuilder for C* manual. The comment for class System in the image describes the messages you can send to invoke these userActions.

GemStone uses this mechanism to build its GemConnect product, which provides access to relational database information from GemStone objects. GemConnect is fully encapsulated and maintained in the GemStone object server. For more information about GemConnect and its capabilities, refer to the *GemConnect Programming Guide*.

If you do not need to manipulate the GemStone objects themselves in your C code, the Foreign Function Interface (FFI) allows you to directly invoke existing C libraries from GemStone Smalltalk, without writing GemBuilder for C interface code. Using the FFI requires that you define the C style structural elements in your GemStone Smalltalk code.

1.9 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.

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

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

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

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

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

Only users who have a UserProfile can log on to the system. For more about UserProfiles, see the *System Administration Guide for GemStone/S 64 Bit* .

See Chapter 8, "Object Security and Authorization."

1.10 Services To Manage the GemStone Repository

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

GemStone provides services that can:

- Support flexible backup and restore procedures.
- Recover from hardware and network failures.
- Perform object recovery when needed.
- Tune the object server to provide high transaction rates.
- Accommodate the addition of new machines and processors without recoding the system.
- Make controlled changes to the definition of the business and application objects in the system.

This manual provides information about programmatical techniques that can be used to optimize your GemStone environment for system administration. Actual system administration and management processes are discussed in the *System Administration Guide for GemStone/S 64 Bit* .

Chapter

2 Programming With GemStone

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

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

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

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

2.1 The GemStone Programming Model

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

Server-Based Classes, Methods, and Objects

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

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

- Handling data
- Processing collections and queries
- Managing the system
- Managing user accounts

The GemStone class library does not provide a user interface. User interface functionality is provided in client Smalltalk products.

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

Client and Server Interfaces

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

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

Topaz
Topaz is a GemStone programming environment that provides 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.

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.

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.

For more information about the GemBuilder and Topaz products, see their respective user manuals. UserActions are discussed in the *GemBuilder for C* manual.

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 session can have only one linked application.

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

2.2 GemStone Smalltalk

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

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

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

Language Extensions

To facilitate your work with persistent objects and large collections, GemStone Smalltalk extends standard Smalltalk in several ways.

Query Syntax

Enterprise applications need to support efficient searching over collections to find all objects that match some specified criteria. Each collection class in GemStone Smalltalk provides methods for iterating over its contents and allowing any kind

of complex operation to be performed on each element. All collection classes understand the messages `select:`, `reject:`, and `detect:`.

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

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

Auto-Growing Collections

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

Class Library Differences

Also to facilitate your work with persistent objects and large collections, GemStone Smalltalk changes the standard Smalltalk class library in several ways.

No User Interface

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

Different File Access

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

Different C Callouts

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

Class Library Extensions

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

More Collection Classes

GemStone Smalltalk provides a number of specialized Collection classes, such as the KeyValueDictionary classes, that have been optimized to improve application speed and support scaling capability. For a full discussion of these classes, see Chapter 5, "Collection and Stream Classes".

Reduced-Conflict Classes

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

User Account and Security Classes

UserProfile is used by GemStone in conjunction with information GemStone gathers during each session to provide a range of security and authorization services, including login authorization, memory and file protection, secondary storage management, location transparency, logical name translation, and coordination of resource use by concurrent users. This manual discusses how UserProfile is used by GemStone during a session. The *System Administration Guide for GemStone/S 64 Bit* contains procedures for creating and maintaining UserProfiles.

Instances of GsObjectSecurityPolicy are used to control ownership of and access to objects. With security policies, you can abstractly group objects, specify who owns the objects, specify who can read them, and specify who can write them. This manual provides a full discussion of GsObjectSecurityPolicies in the Security chapter.

System Management Classes

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

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

Implementation of these classes is discussed in this manual. All of these classes are described in detail in their respective comments in the image.

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 file out objects and file them in to another repository. For more information about the process, see See “File In and File Out” on page 218, or read the description of the PassiveObject class in the image.

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.

For more about this, see Chapter 11, "Signals and Notifiers".

DbTransience

GemStone Smalltalk classes can be DbTransient, meaning their instance variables are not stored to disk. This is useful when your object structure includes classes containing session state such as Semaphores.

2.3 Process Architecture

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

Gem Process

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

Stone Process

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

communicate with the Stone through interprocess channels. The Stone performs the following tasks:

- Coordinates commit processing.
- Coordinates lock acquisition.
- Allocates object IDs.
- Allocates object Pages.
- Writes transaction logs.

Shared Object Cache

The shared object cache provides efficient retrieval of objects from disk, and the ability for multiple Gems to access the same object. A cache is started on each machine that runs a Stone monitor, Gem session process, or linked application.

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

Garbage Collection (GcGem) Processes

The garbage collection (GcGem) processes identify and dynamically reclaim space used by unreferenced objects. The GcGem processes also dynamically defragment the repository while maintaining requested object clustering.

- The *Admin GcGem* is a Gem server process that is dedicated to performing the administrative garbage collection tasks under supervision of the Stone. Each repository can have up to one Admin GcGem process running.
- The *Reclaim GcGems* perform the actual page reclaim operations. On a running GemStone system, there may be between 0 and *n* Reclaim GcGems present, where *n* is the number of extents in the repository.

For details about GemStone garbage collection, see the *System Administration Guide for GemStone/S 64 Bit* .

Extents and Repositories

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

Transaction Log

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

NetLDI

In a distributed system, each machine that runs a Stone monitor, Gem session process, or linked application, must have its own network server process, known as a NetLDI (Network Long Distance Information). A NetLDI is also required if any RPC ("remote") Gem is used, even if all processes are on the same host.

A NetLDI reports the location of GemStone services on its machine to remote processes that must connect to those services. The NetLDI also spawns other GemStone processes on request.

Login Dynamics

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

You can log into GemStone through any of its interfaces: GemBuilder for Smalltalk, GemBuilder for C, GemBuilder for Java, or Topaz. Whichever interface you use, GemStone requires the presentation of a *user ID* (a name or some other identifying string) and a password. If the user ID and password pair match the user ID and password pair of someone authorized to use the system, GemStone permits interaction to proceed; if not, GemStone severs the logical connection.

The system administrator (or a user with equivalent privileges) assigns each GemStone user an instance of class UserProfile, which contains, among other information, the user ID and password. GemStone uses the UserProfile to establish

logical names and default locations, resolve references to system objects, and perform similar tasks. The system administrator gives each new UserProfile appropriate customized rights, and stores it with a set of all other UserProfiles in a set called AllUsers.

You can obtain your own UserProfile by sending a message to System. Class UserProfile defines protocol for obtaining information about default names, privileges, and so forth. This manual provides examples of how UserProfile is used in GemStone applications. For more information about class UserProfile, see the comments in the image. For instructions about creating and maintaining UserProfiles, see the *System Administration Guide for GemStone/S 64 Bit* .

The GemStone system administrator can also configure a GemStone system to monitor failures to log in, to note repeated login attempts, and to disable a user's account after a number of failed attempts to log into the system through that account. The *System Administration Guide for GemStone/S 64 Bit* describes these procedures in greater detail.

Chapter

3

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.

Instance, Class and Other Variables
explains GemStone class implementation formats and other ways classes can store data.

Object Invariance
describes how to make objects invariant.

Non-Persistent Classes and DbTransient
explains how classes can be defined so that their instances or instance variables are not stored in the repository.

3.1 Subclass Creation

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

Example 3.1

```
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.

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.

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. The actual limit for the number of dynamic instance variables is:

```
(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 is persisted in the repository like any other instance variable data. 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 3.2.

Example 3.2

```
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 9.

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

```
Animal description: 'Class describing Animal, created for
the Programmers 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 52 for details. This option cannot be used in combination with `#instanceNonPersistent` 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 `#instanceNonPersistent` or `#dbTransient`
`#instancesNonPersistent`	See page 51 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`.
`#subclassesDisallowed`	No subclasses of the newly created class are permitted.
`#traversalByCallback`	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.

3.2 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 3.3

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

3.3 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 (`#instanceInvariant`).

As discussed in Chapter 5, 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 (`#instanceInvariant`).

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 from the repository, 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 294. Also see Chapter 10 of the *GemStone/S 64 Bit 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.

To determine if an class's instances are DbTransient, send:

```
aClass instancesDbTransient
```

Chapter

4 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.

4.1 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).

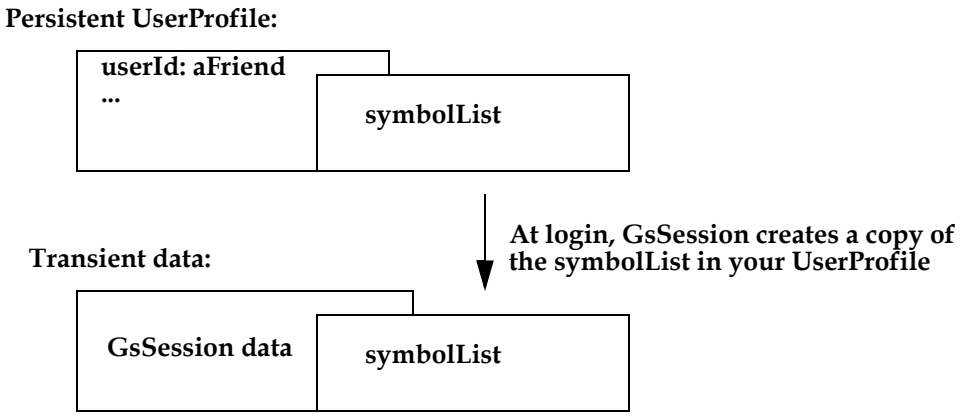
If GemStone cannot find a match for a name in one of these areas, you are given an error message.

4.2 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 4.1.

Figure 4.1 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 4.1

```
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 4.2 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 4.2

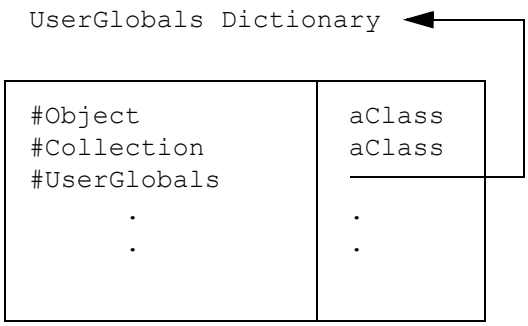
```
topaz 1> printit
UserGlobals keys
%
a SymbolSet
  ...
  #1 GcUser
  #2 UserGlobals
  #3 GsPackagePolicy Current
  #4 PackageLibrary
  ...
```

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 *GemStone/S 64 Bit 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 4.2 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 4.2 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 GemStone/S 64 Bit 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 4.3

```
| 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 4.3, 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 4.4

```
| 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.
%

"Create a new object named MyCollection in UserGlobals dictionary "
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

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

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

Example 4.5

```
System myUserProfile removeDictionaryAt: 1
```

Updating Symbol Lists

There are many ways that the current session's symbol list can get out of sync with the UserProfile symbol list. As some of the examples in this chapter show, updates can made to the current session 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 4.6, 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 4.6

```
"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 4.7

```
topaz 1> printit
"Which symbol dictionary defines the object 'Bag'?"
System myUserProfile symbolResolutionOf: #Bag
4 Globals
    Bag  Bag
```

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

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

Example 4.8 uses `dictionaryAndSymbolOf:` to find out which dictionary in the symbol list stores a reference to class DateTime.

Example 4.8

```
| anArray myUserPro |
"Get the UserProfile"
myUserPro := System myUserProfile.

"Find the Dictionary containing DateTime"
anArray := myUserPro dictionaryAndSymbolOf: DateTime.
anArray at: 1.
aSymbolDictionary

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

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

4.3 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 GemStone/S 64 Bit 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

5 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.

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

Another instance creation message, `new:` *anInteger*, causes many Collection subclass to create an instance with *anInteger* nil 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 also 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:

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

Example 5.1 uses both of these methods to add elements to an instance of Collection's subclass IdentitySet.

Example 5.1

```
| potpourri |
potpourri := IdentitySet new.
UserGlobals at: #Potpourri put: potpourri.
Potpourri add: 'a string of characters'; add: 0.0035;
      add: #aSymbol.
Potpourri addAll: #(#flotsam #jetsam #salvage).
Potpourri
```

IdentitySet is a simple kind of collection, so adding elements is straightforward. Other Collection classes override these methods in order to control access to elements or to enforce an ordering scheme. Still other subclasses of Collection provide additional methods that add an element at a numbered positions or based on an arbitrary key.

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:

Example 5.2

```
UserGlobals
      at: #Virtues
```

```
      put: { 'humility' . 'generosity' . 'veracity' .
'continence' . 'patience' }.
```

```
anArray( 'humility', 'generosity', 'veracity', 'continence',
'patience')
```

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

Example 5.3

```
| aString |
aString := String new. "Make a new, empty String."
"Append a virtue, followed by a space, to the new String"
(Virtues sortAscending) do: [:aVirtue |
      aString := aString , ' ' , aVirtue].
^ aString

' continence generosity humility patience veracity'
```

In 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 during enumeration.

Sorting

You are likely at some point to want to present the contents of your Collection in a sorted order. 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 <=. Messages such as `sortAscending` and `sortDescending` can be sent to any collection that contain only these types of objects. For example:

Example 5.4

```
(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 5.5

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

Employee compileAccessingMethodsFor:
      #('firstName' 'lastName' 'job' 'age' 'address').
%

"Make some Employees and store them in a AllEmployees."
| Lee Kay Al myEmployees |
Lee := (Employee new) firstName: 'Lee'; lastName: 'Smith';
   job: #librarian; age: 40; address: '999 W. West'.
Kay := (Employee new) firstName: 'Kay'; lastName: 'Adams';
   job: #clerk; age: 24; address: '540 E. Sixth'.
Al := (Employee new) firstName: 'Al'; lastName: 'Jones';
```

```
   job: #busdriver; age: 40; address: '221 S. Main'.

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 5.6

```
| 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;
                                add: Character 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 5.7

```
| returnArray tempString |
tempString := String new.
returnArray := AllEmployees sortWith: #('age' 'Ascending'
                                'lastName' 'Descending').
returnArray do: [:i | tempString add: (i age asString);
                                add: ' '; add: i lastName;
                                add: Character 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 81. 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 6. 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 114.

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
```

Sort Ordering and Collation

While the sort ordering of numbers is obvious, Strings are another story. Different languages may sort the same set of strings differently, according to the particular rules in that language. The sort order is called the collation sequence. GemStone, by default, sorts according to the Default Unicode Collation Element Table (DUCET). GemStone allows you to customize collation order. For details, see Appendix F of the *GemStone/S 64 Bit System Administration Guide*.

Changing the collation is risky, since anything that depends on collation, such as indexes, may become unusable if collation changes. Collation in GemStone is the same for all users on a repository, and can only be changed by SystemUser.

The Character Data Tables that control collation also represent the relationships between lowercase and uppercase letters. Case-insensitive searches depend on how this is configured.

5.2 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. Many of the messages that dictionaries understand are specialized for referring to either the key or the value portions of their 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 293.

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 *GemStone/S 64 Bit 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. The SequenceableCollection classes differ from one another mainly in their literal representations, the kinds of elements they store, and the kinds of changes they permit you to make to their instances.

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. In addition, there are several additional methods for adding objects to its instances at particular locations.

`addLast:` does the same thing as `add:`. The message `insertAll:` *aSequenceableCollection* `at:` *anIndex* inserts the elements of a separate SequenceableCollection into the receiver at *anIndex*.

You can remove one or more objects from a SequenceableCollection. Options are remove the first or last element, or removing 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.

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

Copying SequenceableCollection

SequenceableCollection understands three copying messages—one that returns a sequence of the receiver's elements as a new collection, one that copies a sequence of the receiver's elements into an existing SequenceableCollection, and a third message that copies elements from one SequenceableCollection into another without faulting the contents into memory.

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

Example 5.8

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

Example 5.9 copies two elements of an array into a different array, overwriting the target array's original contents:

Example 5.9

```
| numericArray |
numericArray := Array new add: 55; add: 66;
                          add: 77; add: 88; yourself.
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.

Arrays

As you have seen in previous examples, instances of Array and of its subclasses contain elements that you can address with integer keys that describe the positions of Array elements.

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

Creating Arrays

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

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

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

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

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.

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.

Example 5.10

```
| 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 may be 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.

Strings

CharacterCollection expands the protocol inherited from SequenceableCollection to include messages specialized for comparing, searching, concatenating, and changing the case of character sequences.

CharacterCollection is an abstract superclass for String, and the related classes DoubleByteString and QuadByteString. Instances of these subclasses of CharacterCollection and their subclasses are *byte objects*. A byte object has two important practical implications:

- You cannot create a subclass that has named instance variables.

- When you use `new:` to create an instance of a kind of String, GemStone Smalltalk sets each slot of the new instance's indexed instance variables to the Character with value 0 (the null character).

The elements of a String are Characters. 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 converted to a DoubleByteString or QuadByteString as necessary. The protocol is unchanged, so access by index will return the Character at the given index, regardless of how many bytes the String requires.

The following protocol is understood by String, DoubleByteString and QuadByteString. In this discussion, we'll use the term String to mean any of these classes.

Creating Strings

You have already seen many strings created as literals. Strings created as literals are invariant —they cannot be modified later.

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

Example 5.11

```
| 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:

Example 5.12

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

Searching and Pattern-Matching Strings

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. These methods perform both case-sensitive and case-insensitive search and compare. Table 5.1 describes those messages for case-insensitive strings, Table 5.2 describes those messages for case-sensitive strings. This list is not complete; see the image for other methods that can be used.

Table 5.1 String Case-Insensitive Search Protocol

at: *anIndex* equalsNoCase: *aCharCollection*	Returns true if *aCharCollection* is contained in the receiver, starting at *anIndex*. Returns false otherwise.
findPatternNoCase: *aPattern* startingAt: *anIndex*	Searches the receiver, beginning at *anIndex*, for a substring that matches *aPattern*. If a matching substring is found, returns the index of the first character of the substring. Returns zero (0) otherwise.
findStringNoCase: *subString* startingAt: *anIndex*	If the receiver contains *subString* beginning at some point at or after *anIndex*, returns the index at which *subString* begins. Returns zero (0) otherwise.
includesString: *aCharCollection*	Returns true if *aCharCollection* is contained in the receiver. Returns false otherwise.

Table 5.2 String Case-Sensitive Search Protocol

at: *anIndex* equals: *aCharCollection*	Returns true if *aCharCollection* is contained in the receiver, starting at *anIndex*. Returns false otherwise.
findPattern: *aPattern* startingAt: *anIndex*	Searches the receiver, beginning at *anIndex*, for a substring that matches *aPattern*. If a matching substring is found, returns the index of the first character of the substring. Returns zero (0) otherwise.
findString: *subString* startingAt: *anIndex*	If the receiver contains *subString* beginning at some point at or after *anIndex*, returns the index at which *subString* begins. Returns zero (0) otherwise.
matchPattern: *aPattern*	Returns true if the receiver matches *aPattern*, false if it doesn't. An exact match is required.

Pattern matching arguments (*aPattern*) consist of an Array containing zero or more Strings plus zero or more occurrences of the special wildcard characters $* or $?.

The character $? matches any single character in the receiver, and $* matches any sequence of characters in the receiver

Example 5.13 shows the use of wildcard characters in pattern matching.

Example 5.13

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

This example returns true because the character $* is interpreted as "any sequence of characters." Similarly, Example 5.14 returns the index at which a sequence of characters beginning and ending with $r occurs in the receiver.

Example 5.14

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

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

Example 5.15

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

Comparing Strings

The Class String supports case-insensitive String comparisons, with additional messages for case sensitivity where that behavior is desired. The following behavior is provided in String:

`=`	compare case-sensitive
`isEquivalent:`	compare case-insensitive
`equalsNoCase:`	compare case-insensitive

The methods <, <=, >, and >= first perform a case-insensitive comparison of the receiver and argument; if they are found to be equal — that is, the only difference is case — then they are compared according to the current collation, which specifies uppercase comes before lowercase as AÀÁ...aàá...

For example, the following message:

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

results in:

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

The following methods take a user-defined collating sequence. If you need results ordered by ASCII, for example, use these methods with the table provided in `Globals at: #AsciiCollatingTable`. The methods can be used in sort blocks of SortedCollections, but they are not usable by implementations of `sortAscending`: or `sortDescending`:.

```
lessThan:collatingTable:

lessThanOrEqual:collatingTable:

greaterThan:collatingTable:

greaterThanOrEqual:collatingTable:
```

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:

Example 5.16

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

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

string. Note that you cannot start with a literal string, since a literal string is invariant. For example:

Example 5.17

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

Converting Strings to Other Kinds of Objects

CharacterCollection and its subclasses define messages that let you convert a string to an upper- or lowercase string, to a symbol, to various numeric kinds, or perform other conversions. See the image for details on all the conversion methods available. 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:

Example 5.18

```
'ABCDE' asLowercase
abcde

'abcde' asUppercase
ABCDE

'abcde' asSymbol
abcde

'15' asFloat
1.5000000000000000E+01
```

String Equality and Identity

Literal and nonliteral InvariantStrings and Strings behave differently in identity comparisons. Each nonliteral String (created, for example, with `new`, `withAll:`, or `asString`) has a unique identity. That is, two Strings that are equal are not necessarily identical. See Example 5.19.

Example 5.19

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

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

Example 5.20

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

This distinction can become significant in building sets. Because a set does not accept more than one element with the same identity, if you add both *litString1* and *litString2* to the same set, the set will contain only one instance of `'abc'`. You can, however, store both *nonlitString1* and *nonlitString2* in a single set.

Symbols

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

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.

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 never been previously defined. Existing Symbols cannot be modified.

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 6, "Querying," describes the querying/sorting functions in detail. The following section describes the protocol implemented in UnorderedCollection's subclasses.

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. ABag 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 5.21

```
| 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 5.22

```
| 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 5.23

```
| 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 5.23 works correctly because the SmallInteger 2 has a unique identity throughout the system. By way of contrast, consider Example 5.24.

Example 5.24

```
| 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 5.25

```
| 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.

Class IdentitySet

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

5.3 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.

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:

```
Stream class >> installPortableStreamImplementation.
```

To install the legacy version:

```
Stream class >> installLegacyStreamImplementation
```

Chapter

6 Querying

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

Relations
reviews the concept of relations.

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

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

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

Sorting and Indexing
describes protocol for sorting collections efficiently.

6.1 Relations

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

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

Figure 6.1 Employee Relation

Employees

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

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

To make it easy to retrieve values from a record, you can define selectors in class Employee so that an instance of Employee returns the value of its *name* instance variable when it receives the message `name`, the value of its *age* variable when it receives the message `age`, and so on. The discussion of class IdentitySet in Chapter 5, "Collection and Stream Classes," describes one way to develop this Employee class.

As Chapter 5 also explains, you can use Collection's searching protocol to search for a record (element) containing a particular field (instance variable) value.

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

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

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

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

What You Need To Know

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

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

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

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

6.2 Selection Blocks and Selection

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

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

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

Example 6.1

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

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

Example 6.2

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

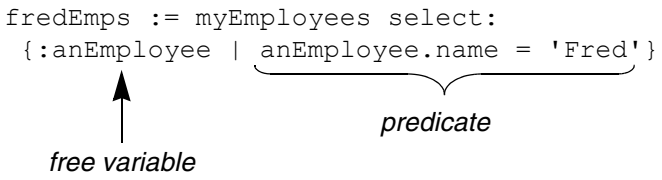
While square brackets `[]` delimit an ordinary block, curly braces `{}` delimit a selection block; Otherwise, the two statements look the same. A query using a selection block also returns the same results as if the selection block predicate had been treated as a series of message expressions. However, some special restrictions apply to the query language.

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

Selection Block Predicates and Free Variables

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

Figure 6.2 Anatomy of a Selection Block



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

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

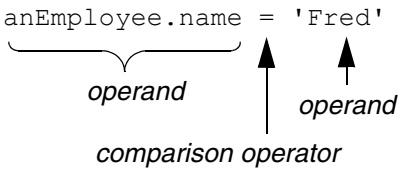
The method for `select:` gathers into the collection fredEmps each element whose *name* value makes the predicate true.

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

Predicate Terms

A *term* is a Boolean expression containing an operand and usually a comparison operator followed by another operand, as shown in Figure 6.3:

Figure 6.3 Anatomy of a Selection Block Predicate Term



Predicate Operands

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

If a path points to objects within elements of `select:`'s receiver (as does *anEmployee.name*), then each variable in the path must be a valid instance variable name for the receiver and its elements. In this case, *anEmployee.name* is acceptable because the receiver holds employees and class Employee defines the instance variable *name*.

Predicate Operators

Table 6.1 lists the comparison operators used in a selection block predicate:

Table 6.1 Comparison Operators Allowed in a Selection Block

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

No other operators are permitted in a selection block.

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

Example 6.3

```
| aTime |
aTime := Time now.
myEmployees select: {:i | aTime == i.name}
```

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

Predicate Operators and User-Defined Classes

If you need to, you can redefine the equality operators =, <=, <, >=, > in classes that you have created. In that case, the operands compared using these operators need not be of the same class. If you have created a class and redefined its equality operators, you can compare instances of that class with any other class that make sense for your application. For further details, see "Redefined Comparison Messages in Selection Blocks" on page 102.

Bear in mind that, in general, indexing is significantly more efficient when the indexed objects are instances of certain GemStone Smalltalk kernel classes. Instances of these classes are compared for equality as follows:

Nil < Symbol < String < DoubleByteSymbol < DoubleByteString < Boolean < Character < Time < Date < subclasses of Number

In less-than or greater-than comparisons, your queries must accommodate for heterogeneous values.

Conjunction of Predicate Terms

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

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

Example 6.4

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

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

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

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

Example 6.5 shows how you can get a collection of all employees named either Fred or Ted.

Example 6.5

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

Limits on String Comparisons

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

Redefined Comparison Messages in Selection Blocks

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

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

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

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

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

- a <= b if a < b or a = b.
- If a = b, then b = a.
- If a < b, then b > a.
- If a >= b, then b <= a.

You must obey one other rule as well: objects that are equal to each other must have equal hash values. Therefore, if you redefine =, you must also redefine the method `hash` so that dictionaries will behave in a consistent and logical manner.

Suppose that you define the class Soldier as follows:

Example 6.6

```
Object subclass: #Soldier
      instVarNames: #( rank )
      classVars:  #( #Ranks )
      classInstVars: #()
      poolDictionaries: #()
      inDictionary: UserGlobals
```

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

Example 6.7

```
Soldier compileAccessingMethodsFor: Soldier instVarNames .

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

We then initialize the class by executing the expression:

```
Soldier initialize
```

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

Example 6.8

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

We now create instances of Soldier having each possible rank, naming them aLieutenant and so on. We also create an instance of Soldier without any rank, and name it aPrivate. See Example 6.9.

Example 6.9

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

We can now execute expressions of the form:

Example 6.10

```
aLieutenant < aMajor
true

aCaptain < aLieutenant
false
```

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

Example 6.11

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

Changing the Ordering of Instances

Once you redefine the equality operators for a given class and create instances of that class, your instances may not remain the same forever. For example, the

soldiers we created in Example 6.9 may not all stay the same rank for their entire careers. Some may be promoted; others may be demoted. If an instance of Soldier changes its ordering relative to the other instances, you must manually update the equality index in which it participates. Because you have redefined the equality operators, GemStone has no way of determining how to update the index automatically, as it will when you use the system-supplied equality operators.

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

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

   ```
   anArray := self removeObjectFromBtrees
   ```

   The method `removeObjectFromBtrees` returns an array that you will need later within the method. Therefore you must assign the result to some variable—`anArray` in the example above.
3. After the code that changes the ordering of the instance, include a line such as the following:

   ```
   self addObjectToBtreesWithValues: anArray
   ```

4. Once you have performed the method `removeObjectFromBtrees`, do not commit the transaction unless the `addObjectToBtreesWithValues:` method successfully completes.

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

CAUTION

Failing to include these lines can corrupt the equality index and lead to your application receiving GemStone errors notifying you that certain objects do not exist. You may need to remove all indexes related to a collection in order to fix the problem.

Collections Returned by Selection

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

NOTE

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

```
RcIdentityBag withAll: myBag
RcIdentityBag new addAll: myBag; yourself
```

RcQueue displays comparable behavior; the message `select:` *returns an Array. For further details on class RcIdentityBag, see Chapter 7, Transactions and Concurrency Control.*

The collection returned by a selection query has no index structures. (Indexes are discussed in detail beginning on page 110.) This is because indexes are built on individual instances of unordered collections rather than the classes. If you want to perform indexed selections on the new collection, you must build all of the necessary indexes. A later section, "Duplicating a Collection's Indexes" on page 118, describes a technique for duplicating a collection's indexes in a new instance.

Streams Returned by Selection

The result of a selection block can be returned as a stream instead of a collection. Returning the result as a stream is faster. If you are not sure that your query is precisely the right one, using a stream allows you to test the results with minimal overhead. It also allows you to iterate over the elements that are returned in the order defined by the index.

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

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

3. Each object in the result must be put into the collection.

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

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

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

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

Example 6.12

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

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

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

- It takes a single predicate only; no conjunction of predicate terms is allowed.
- The collection you are searching must have an equality index on the path specified in the predicate. (Creating equality indexes is discussed in the section "Indexing for Faster Access" on page 110.)
- The predicate can contain only one path.

For example, the predicate in Example 6.13 compares the result of one path with the result of another and therefore cannot be used with `selectAsStream:`

Example 6.13

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

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

myEmployees selectAsStream:
     { :emp | emp.age > emp.lengthOfService }
%
------------------------------------------------------
GemStone: Error         Nonfatal
 The predicate for selectAsStream was invalid.
Error Category: [GemStone] Number: 2313 Arg Count: 1
```

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

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

Additional Query Protocol

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

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

Example 6.14

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

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

Example 6.15

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

For UnorderedCollections (NSCs), there is no telling which element will be returned when there are several qualified candidates. If no elements are qualified, `detect:` issues an error notification and the interpreter halts.

If you don't want the interpreter to halt in the event of a fruitless search, use `detect:` *aBlock* `ifNone:` *exceptionBlock*.

6.3 Indexing for Faster Access

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

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

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

Identity Indexes

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

Example 6.16

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

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

Creating Identity Indexes

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

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

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

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

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, the `createIdentityIndexOn:` method is terminated with an error.

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

Creating Indexes on Large Collections

For large collections, it may take a long time to create an index in a single transaction. By breaking the index creation into multiple, smaller transactions, the overall time required to build the index is shorter.

What's more, creating indexes can consume a significant amount of temporary object memory, which can lead to out-of-memory conditions.

For these reasons, you may choose to commit your work to the repository incrementally during index creation. For details, see "Indexes and Transactions" on page 114.

Equality Indexes

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

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

Example 6.17

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

As explained in Table 6.1 (on page 100), equality queries use the related comparison operators =, <, <=, >, and >=.

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

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

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

Creating Equality Indexes

The technique for creating equality indexes is similar to the technique for creating identity indexes, with one significant difference: you must specify the class of the final element of the path:

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

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

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

myEmployees createEqualityIndexOn: 'address'
      withLastElementClass: Address.

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

Creating Reduced Conflict Equality Indexes

If you are creating an index on an reduced-conflict (RC) collection, such as RcIdentityBag, you may benefit from creating RC equality indexes rather than plain equality indexes. This will avoid some transaction conflicts on the indexing structures themselves, which may happen even if there are no conflicts between modifications to the collection itself (for details on this, see "Indexes and Concurrency Control" on page 132).

The protocol for creating reduced conflict equality indexes is the similar to equality indexes:

```
createRcEqualityIndexOn: aPath withLastElementClass: aClass
```

IdentityIndexes always use internal structures that are reduced conflict.

Creating Indexes on Large Collections

For large collections, it may take a long time to create an index in a single transaction. By breaking the index creation into multiple, smaller transactions, the overall time required to build the index is shorter.

What's more, creating indexes can consume a significant amount of temporary object memory, which can lead to out-of-memory conditions.

For these reasons, you may choose to commit your work to the repository incrementally during index creation. For details, see "Indexes and Transactions" on page 114.

Automatic Identity Indexing

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

Implicit Indexes

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

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

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

6.4 Managing Indexes

After creating indexes, you may at times wish 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
```

Indexes and Transactions

Modifying an object that participates in an index on some collection can, under certain circumstances, write certain objects built and maintained internally by GemStone as part of the indexing mechanism. Chapter 7, "Transactions and Concurrency Control," explains the significance of your writing an object.

Committing Your Work Incrementally

The class IndexManager controls the transactional behavior of index creation and removal. IndexManager provides methods that allow you to commit your work to the repository incrementally during index creation (or removal). As mentioned earlier, this approach enables you to avoid out-of-memory conditions, while significantly reducing the overall time required to build the index.

When you send the following message:

```
IndexManager autoCommit: true
```

the current transaction is committed during indexing whenever the current session receives a signal indicating temporary object memory is almost full, or when either of these thresholds is reached:

- dirtyObjectCommitThreshold — When the number of objects that have been modified (that is, have become "dirty") during the current transaction exceeds this threshold, the transaction is committed. The default is SmallInteger maximum value, which means this limit is effectively disabled.

  To change this threshold, send the message:

  `IndexManager >> dirtyObjectCommitThreshold:` *anInt*

- percentTempObjSpaceCommitThreshold — When the percentage of temporary object memory in use reaches this threshold, the transaction is committed. (When this value is nil, the threshold is ignored.) The default is 60. To change this threshold, send the message:

  `IndexManager >> percentTempObjSpaceCommitThreshold:` *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.

Inquiring About Indexes

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

- `equalityIndexedPaths` and `identityIndexedPaths`

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

For example, the following expression returns the paths into myEmployees that bear equality indexes:

```
myEmployees equalityIndexedPaths
```

- `kindsOfIndexOn:` *aPathNameString*

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

- `equalityIndexedPathsAndConstraints`

  Returns an array in which the odd-numbered elements are the elements of the path, and the even-numbered elements are the constraints 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.

- `getAllIndexes`

  Returns a collection of all indexes on all UnorderedCollections in the repository.

The following sections describe several practical uses for these messages.

Removing Indexes

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

- `removeEqualityIndexOn:` *aPathString*

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

- `removeIdentityIndexOn:` *aPathString*

  Removes identity indexes. If the path specified does not exist, the method returns an error. If the path specified was implicitly created, the method

returns the path, but the index is not removed. If the index is successfully removed, the method returns the receiver.

- `removeAllIndexes`

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

The IndexManager provides additional protocol for removing all indexes in the repository.

- `removeAllIndexes`

  Removes all indexes on all UnorderedCollections.

- `removeAllIncompleteIndexes`

  Removes all incomplete indexes on all UnorderedCollections. This method is used when an error occurs during index creation with autoCommit: on, so portions of the index being created have been committed.

Removing Indexes from Large Collections

For large collections, it may take a long time to remove an index in a single transaction. By breaking the index removal into multiple, smaller transactions, the overall time required to remove the index is shorter.

What's more, removing indexes can consume a significant amount of temporary object memory, which can lead to out-of-memory conditions.

For these reasons, you may choose to commit your work to the repository incrementally during index removal. For details, see "Indexes and Transactions" on page 114.

Implicit Index Removal

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

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

Duplicating a Collection's Indexes

As explained on page 107 ("Collections Returned by Selection"), a collection returned by `select:` is devoid of indexing, even when `select:`'s receiver has indexes in place. Fortunately, the index inquiry protocol for UnorderedCollection makes it easy to duplicate the indexes in a new collection. See Example 6.18.

Example 6.18

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

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

1 to: (equalityIndexes size) by: 2 do:
     [ :n | someEmployees createEqualityIndexOn:
                    (equalityIndexes at: n )
                withLastElementClass:
                    (equalityIndexes at: n + 1)].
```

Removing and Re-Creating Indexes

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

When you change the value of an object that participates in an index, GemStone Smalltalk *in most cases* automatically adjusts the indexes to accommodate the new value. When changing the value of any object more complex than a Number or

String, however, you must be especially careful. For more about this, see "Changing the Ordering of Instances" on page 105.

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

Removing Residual Indexes

As stated earlier, you must explicitly remove any indexes that you have created explicitly. If you attempt to dereference an UnorderedCollection on which indexes still exist, those residual indexes will prevent the collection from being successfully garbage-collected, and you will be unable to free up permanent object memory.

With IndexManager autoCommit set to true, commits may occur during index creation. If an error occurs after portions of the index have been created and committed, the unusable partial index must be explicitly removed. The IndexManager defines instance methods to remove incomplete indexes:

`IndexManager current removeAllIncompleteIndexes`

Removes all incomplete indexes on all UnorderedCollections.

`IndexManager current removeAllIncompleteIndexesOn:` *anNSC*

Removes all incomplete indexes on the specified UnorderedCollection.

Indexing and Performance

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

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

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

- the object participates in many indexes.

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

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

Indexing Errors

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

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

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

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.

For details on committing transactions, see Chapter 7.

Auditing Indexes

Indexes should be audited regularly, as part of your regular application maintenance to ensure there are no problems. In the rare case of a problem report, Contact Gemstone Technical Support to determine the cause of the problem. Most often the solution is to remove and rebuild the affected indexes.

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'. 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.

6.5 Sorting and Indexing

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

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

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

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

Chapter 7

Transactions and Concurrency Control

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

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

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

Concurrency Management
introduces optimistic and pessimistic concurrency control.

Controlling Concurrent Access with Locks
discusses the 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.

7.1 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 7.1, 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, the frequency of transactions, and the extent of object sharing), this burden can be trivial or significant.

When you log in to GemStone, a transaction is started for you. Your current transaction exists until you successfully commit the transaction, abort it, or log out. Your view endures for the length of the transaction, unless you explicitly choose to get a new view by *continuing* the transaction. When you obtain a new view of the repository, any new or modified objects that have been committed by other users become visible to you.

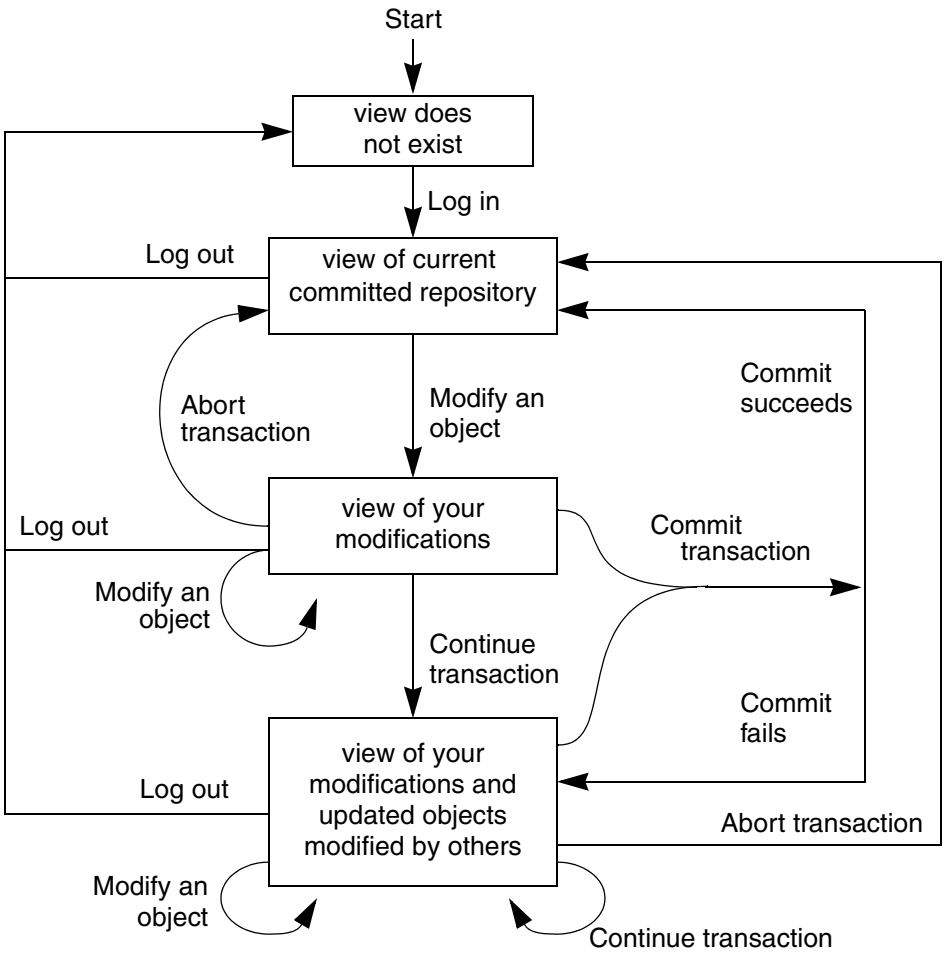
(In manual transaction mode, even though your session may exist outside of a transaction, your view is not updated until you begin a transaction and then commit, abort, or continue it. For details, see "Transaction Modes" on page 128.)

When Should You Commit a Transaction?

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

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

Figure 7.1 View States



Reading and Writing in Transactions

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

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

Operations that fetch information about an object also read the object. In particular, fetching an object's size, class, or 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.

7.2 How GemStone Detects Conflict

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

- An object in your write set is also in the write set of another transaction—a *write-write conflict*. Write-write conflicts can involve only a single object.

- An object in your write set is 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 6, 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.

Transaction Modes

You use GemStone in any of three modes:

- **Automatic transaction mode.** In this mode, GemStone begins a transaction when you log in, and starts a new one after each commit or abort message. In this default mode, you are in a transaction the entire time you are logged into a GemStone session. If the work you are doing requires committing to the repository frequently, you need to use automatic transaction mode, as you cannot make permanent changes to the repository when you are outside a transaction.
- **Manual transaction mode.** In this mode, you can be logged in and outside of a transaction. You explicitly control whether your session can commit. 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.
- **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 determine the transaction mode you are in, print the result of sending the message:

```
System transactionMode
```

Changing Transaction Mode

To change to manual transaction mode, send the message:

```
System transactionMode: #manualBegin
```

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

Beginning a New Transaction in Manual Mode

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

```
System beginTransaction
```

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

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

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

```
System inTransaction
```

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

Committing Transactions

Committing a transaction has two effects:

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

When you tell GemStone to commit your transaction, the object server 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 message `commitTransaction` commits the current transaction:

Example 7.1

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

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

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

```
System transactionConflicts
```

This method returns a symbol dictionary 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 7.1 lists the possible keys for the conflict.

Table 7.1 Transaction Conflict Keys

Key	Meaning

Table 7.1 Transaction Conflict Keys (Continued)

#'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 130). 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.

Aborting Transactions

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

Then you need to *abort* the transaction. This discards all of the modifications from the aborted transaction, and gives you a new view containing the shared, committed objects. Depending on the activities of other users, you can repeat your operations using the new values and commit the new transaction without encountering conflicts.

The message `abortTransaction` discards the modified objects in your view. If you are in automatic transaction mode, this message also begins a new transaction.

Example 7.2

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

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

Aborting a transaction discards any changes you have made to shared objects during the transaction. However, work you have done 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 *GemStone/S 64 Bit 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 reenable 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.

To determine if the current session is in transaction, send the message:

```
System inTransaction
```

7.3 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 7.3

```
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 142.

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 7.4 uses this technique to request a lock repeatedly until the lock becomes available.

Example 7.4

```
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 7.5

```
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 7.6.

Example 7.6

```
| 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 7.7

```
UserGlobals at: #myArray put: Array new;
   at: #myBag put: IdentityBag new.
%

System readLockAll: myArray.
System writeLockAll: myBag.
%
```

The messages in Example 7.7 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 7.8.

Example 7.8

```
classmethod: Dummy
handleDenialOn: deniedObjs
^ deniedObjs
%
classmethod: Dummy
getWriteLocksOn: aCollection
System writeLockAll: aCollection
   ifIncomplete: [:denied :dirty :unused |
      denied isEmpty ifFalse: [self handleDenialOn: denied].
      dirty isEmpty ifFalse: [System abortTransaction] ]
%
System readLockAll: myEmployees
%
Dummy getWriteLocksOn: myEmployees
%
```

Upgrading Locks

On occasion, you might want to *upgrade* a 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 132. 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 7.9 uses `lockOwners:` to build an array of the userIDs of all users whose sessions hold locks on a particular object.

Example 7.9

```
classmethod: Dummy
getNamesOfLockOwnersFor: anObject
| userIDArray sessionArray |
sessionArray := System lockOwners: anObject.
userIDArray := Array new.
sessionArray do:
      [:aSessNum | userIDArray add:
                (System userProfileForSession: aSessNum) userId].
^userIDArray
%

Dummy getNamesOfLockOwnersFor: (myEmployees detect: {:e | e.name =
'Conan' })
%
```

You can test to see whether an object is included in either of the sets of locked objects whose locks are to be released upon the next abort or commit operation. The following statement returns true if *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

7.4 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 7.10

```
!session 1
UserGlobals at: #binCount put: RcCounter new.
System commitTransaction.
%
!session 2
binCount incrementBy: 48.
System commitTransaction.
%
!session 1
binCount incrementBy: 24.
System commitTransaction.
%
!session 3
binCount decrementBy: 144
   ifLessThan: -24
   thenExecute: [^'Not enough widgets to ship today.'].
System commitTransaction.
%
```

RcCounter is not appropriate for all applications—for example, it would not be appropriate to use in an application that keeps track of the amount of money in a shared checking account. If two users of the checking account both tried to withdraw more than half of the balance at the same time, an RcCounter would allow both operations without conflict. Sometimes, however, you need to be warned—for example, of an impending overdraft.

RcIdentityBag

The class RcIdentityBag provides much of the same functionality as IdentityBag, including the expected behavior for `add:`, `remove:`, and related messages. However, no conflict occurs on instances of RcIdentityBag when any 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 132.) You can reduce the risk further by using reduced conflict equality indexes; see "Creating Reduced Conflict Equality Indexes" on page 113. 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 manner in which RcQueues are implemented, reclaiming the storage of objects that have been removed from the queue actually occurs when new objects are added. If a session adds a great many objects to the queue all at once and then does not add any more as other sessions consume the objects, performance can become degraded, particularly from the consumer's point of view. In order to avoid this, the producer can send the message `cleanupMySession` occasionally to the instance of the queue from which the objects are being removed. This causes storage to be reclaimed from obsolete objects.

NOTE

If you subclass and reimplement these methods, build in a check for nils. Because of lazy initialization, the expected subcomponents of the RcQueue may not exist yet.

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

8 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.

8.1 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 authenication 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 4, "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 *GemStone/S 64 Bit 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 *GemStone/S 64 Bit 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.

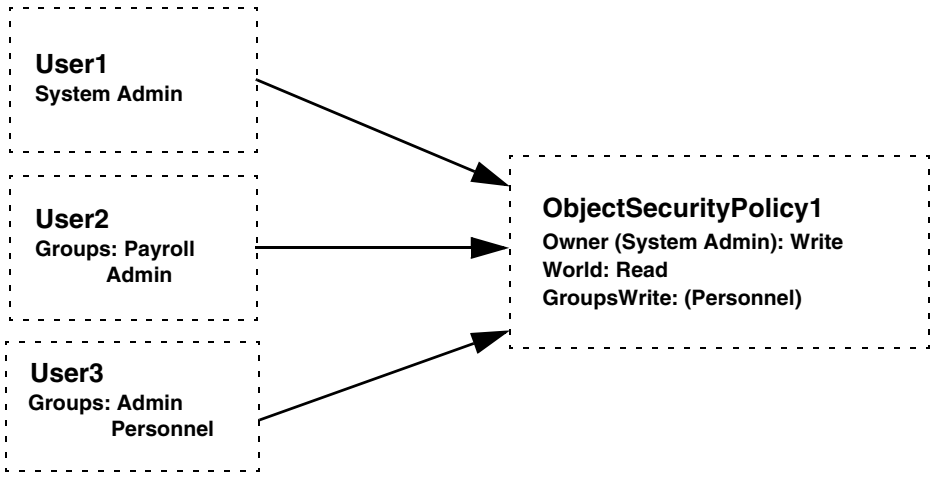
All objects assigned to an security policy have exactly the same protection. That is, if you can read or write one object assigned to a certain policy, you can read or write them all. Each policy is owned by a single user, and all objects assigned to the same security policy have the same owner. Groups 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 8.1:

Figure 8.1 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 4, "Resolving Names and Sharing Objects".

8.2 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 8.1

```
| aPolicy myPolicy |
myPolicy := System myUserProfile
```

```
      defaultObjectSecurityPolicy.
aPolicy := GsObjectSecurityPolicy newInRepository:
      SystemRepository.
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 8.2

```
| myPolicy |
"get default security policy"
myPolicy := System myUserProfile defaultObjectSecurityPolicy.
"compare with current"
myPolicy = System currentObjectSecurityPolicy

true
```

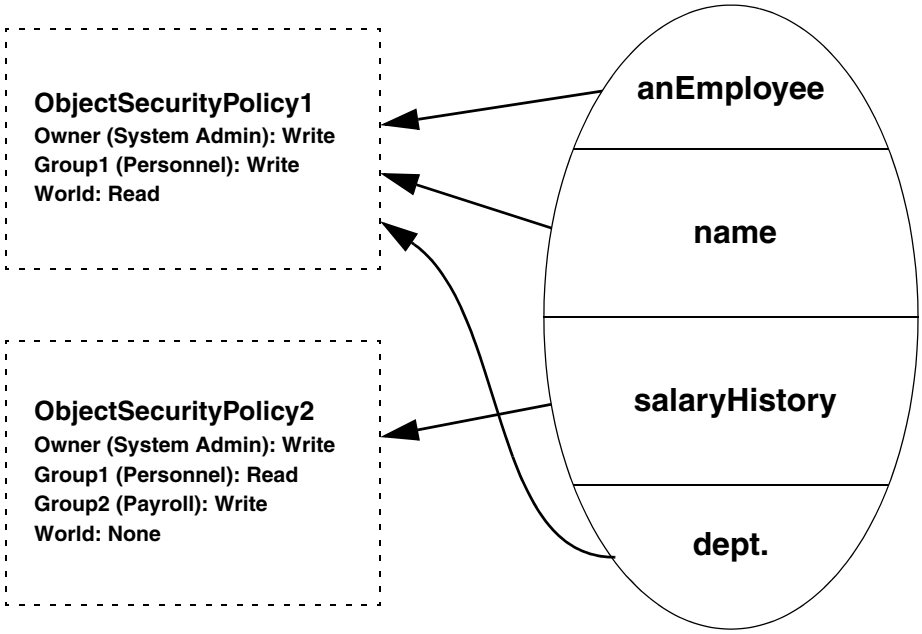
Objects and Security Policies

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 8.2. 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 8.2 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.

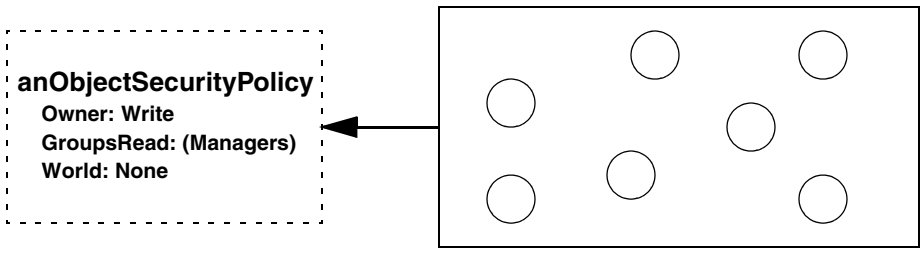
The message `GsObjectSecurityPolicy>>ownerAuthorization:` *anAuthorizationSymbol* is used to set and clear authorization for the owner of the security policy. The message `GsObjectSecurityPolicy>>ownerAuthorization` returns the authorization for the owner of the security policy.

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 8.3, 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 8.3 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 8.3, 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 8.3

```
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 8.4. 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 8.4

```
Array subclass: 'Menagerie'
      instVarNames: #()
      classVars: #()
      classInstVars: #()
      poolDictionaries: {}
      inDictionary: UserGlobals
%
method: Menagerie
objectSecurityPolicy: aPolicy
"Assign the receiver and each component to the given
objectSecurityPolicy."
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 8.5

```
"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:
aUserProfile
```

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 v3.0. 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`.

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.

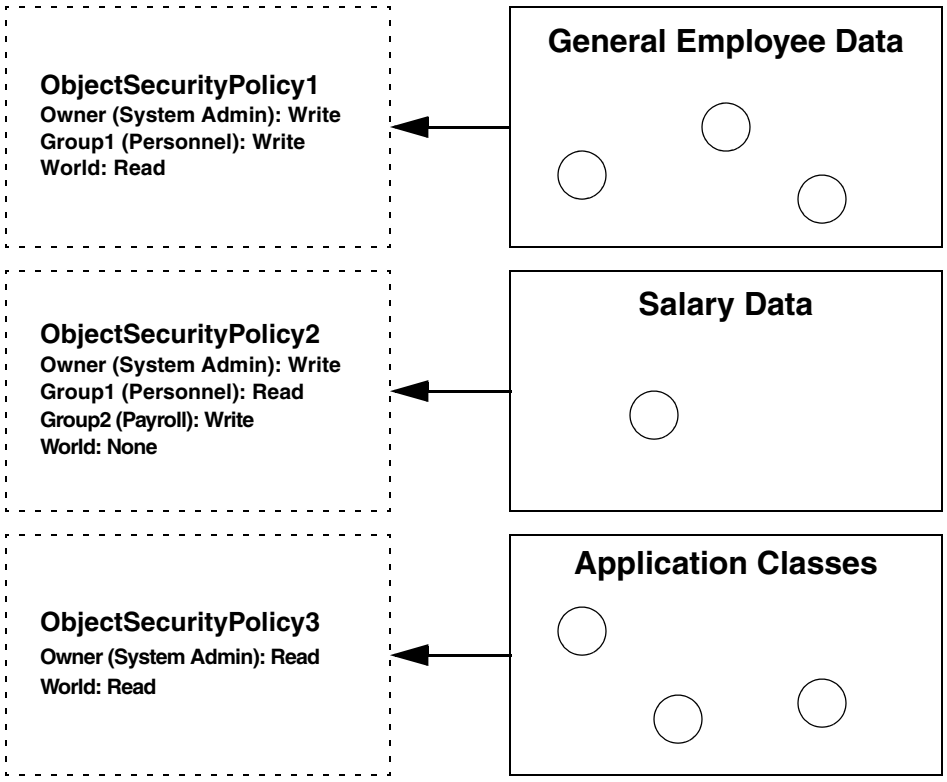
8.3 An Application Example

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 8.4 shows a typical production situation.

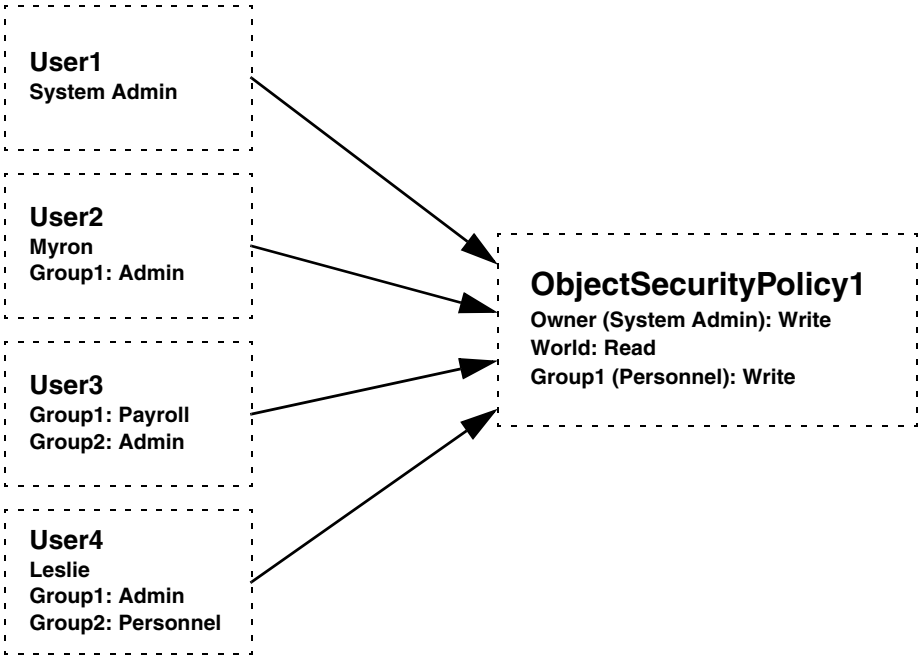
Figure 8.4 Application Objects Assigned to Three Security Policies



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 8.5 and Figure 8.6:

Figure 8.5 User Access to Application ObjectSecurityPolicy1

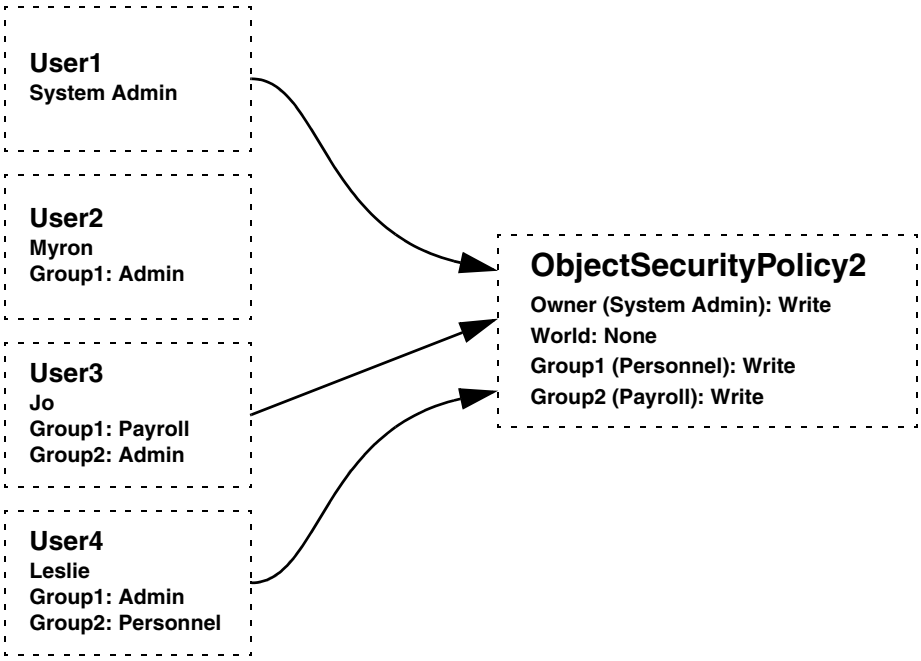


Four users access this application:

- 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 8.6 User Access to Application ObjectSecurityPolicy2

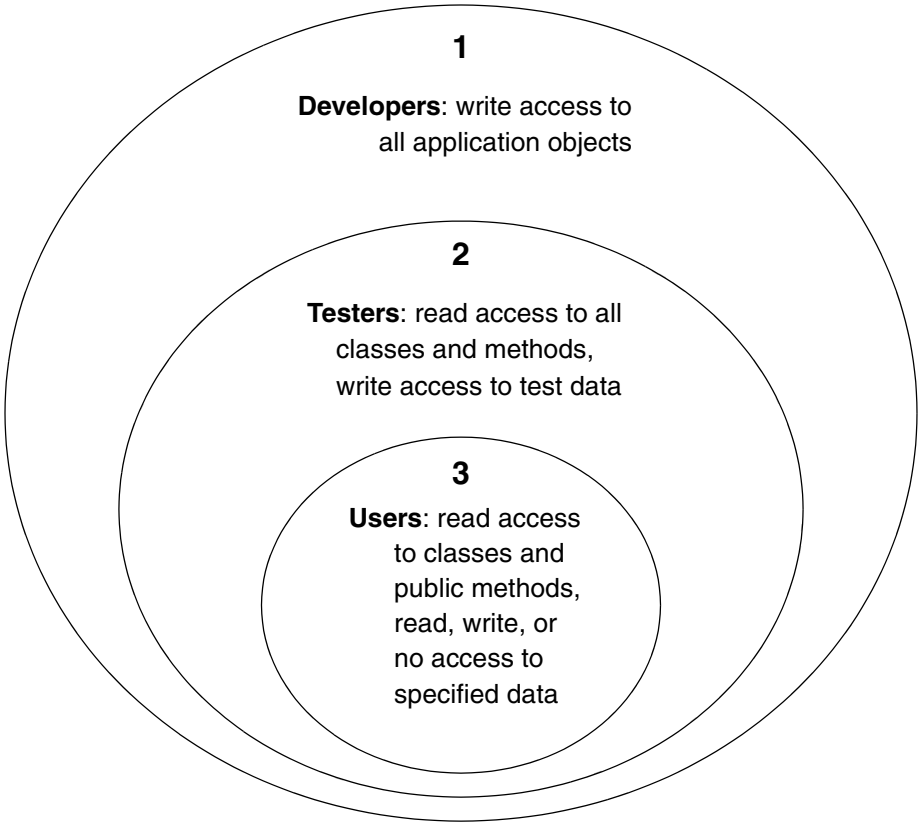


8.4 A Development Example

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 8.7.

Figure 8.7 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 8.1 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 8.1 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 4, "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 8.1 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 8.2. 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 8.2 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 8.1) so you only have to create one group for the two departments. This group, named Personnel, requires write authorization for the objects in Table 8.2.

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 8.3.

Table 8.3 Access to the Last Six Objects Through Owner and World Authorization

	Users						
Objects	**System Admin.**	**Human Resource**	**Employ. Records**	**Payroll**	**Mktg**	**Sales**	**Customer Support**
dateHired	W	W	W	R	N	R	N
salary	W	R	R	W	N	N	N
salesQuarter	W	R	R	R	N	W	N
salesYear	W	R	R	R	N	W	N
vacationDays	W	W	W	N	N	N	N
sickDays	W	W	W	N	N	N	N

write access as owner or no access as world

Two more departments, Human Resources and Employee Records, are already set up to access as the Personnel group. As shown in Table 8.4, 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 8.4 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

read or write access as Personnel group

Table 8.4 Access to the Last Six Objects Through the Personnel Group

	Users						
Objects	**System Admin.**	**Human Resource**	**Employ. Records**	**Payroll**	**Mktg**	**Sales**	**Customer Support**
salesYear	W	R	R	R	N	W	N
vacationDays	W	W	W	N	N	N	N
sickDays	W	W	W	N	N	N	N

read or write access as Personnel group

Now the Payroll and Sales departments still require access to the objects, as shown in Table 8.3. Because these departments' needs don't match anyone else's, they must each have a separate group.

Table 8.5 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 8.8. 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 *GemStone/S 64 Bit System Administration Guide* for implementing that plan.

Setting Up Security Policies for Joint Development

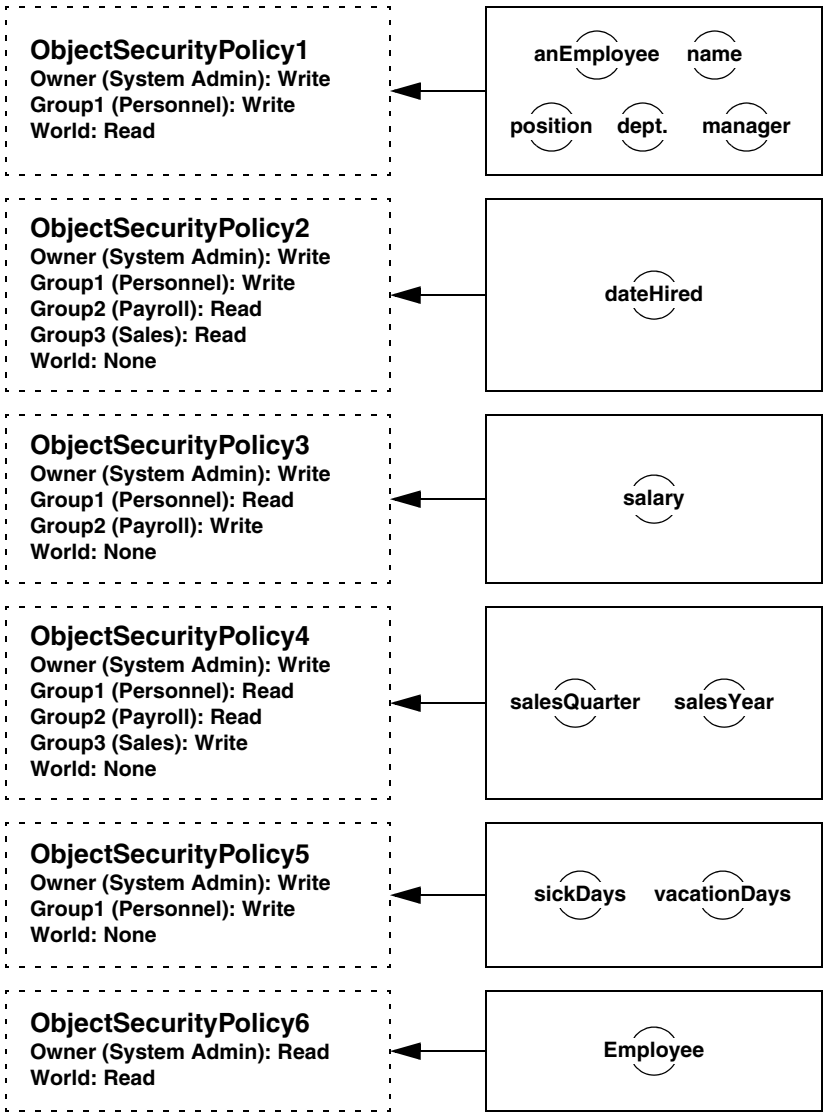
To make joint development possible, you need to set up authorization and references so that all the developers have access to the classes and methods that are being created. Create a new symbol dictionary for the application and put it in everyone's symbol list; make sure it includes references to any 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 8.8 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 4, "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 8.8 on page 184). 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.

8.5 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:

```
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 *GemStone/S 64 Bit 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 newInRepository: SystemRepository.`

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 *GemStone/S 64 Bit System Administration Guide*.

Chapter

9 Class Creation, Versions, and Instance Migration

Few of us can design something perfectly the first time. 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.

9.1 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.

9.2 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 9.1

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

Example 9.2 creates a class named NewAnimal and specifies that the class shares the class history used by the existing class Animal.

Example 9.2

```
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 9.3.

Example 9.3

```
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 quite 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
```

9.3 Migrating Objects

Once you define two or more versions of a class, you may wish to migrate instances of the class from one version to another. Migration in GemStone Smalltalk is a flexible, configurable operation:

- Instances of any class can migrate to any other, as long as they share a class history. The two classes need not be similarly named, or, indeed, have anything else in common.
- Migration can occur whenever you specify.
- Not all instances of a class need to migrate at the same time—you can migrate only certain instances at a time. Other instances need never migrate, if that is appropriate.
- The manner in which values of the old instance variables are used to initialize values of the new instance variables is also under your control. A default mapping mechanism is provided, which you can override if you need to.

Migration Destinations

If you know the appropriate class to which you wish to migrate instances of an older class, you can set a migration destination for the older class. To do so, send a message of the form:

```
OldClass migrateTo: NewClass
```

This message configures the old class to migrate its instances to become instances of the new class, but only when it is instructed to do so. Migration does not occur as a result of sending the above message.

It is not necessary to set a migration destination ahead of time. You can specify the destination class when you decide to migrate instances. It is also possible to set a migration destination, and then migrate the instances of the old class to a completely different class, by specifying a different migration destination in the message that performs the migration.

You can erase the migration destination for a class by sending it the message `cancelMigration`. For example:

```
OldClass cancelMigration
```

If you are in doubt about the migration destination of a class, you can query it with an expression of the form:

```
MyClass migrationDestination
```

The message `migrationDestination` returns the migration destination of the class, or nil if it has none.

Migrating Instances

A number of mechanisms are available to allow you to migrate one instance, or a specified set of instances, to either the migration destination, or to an alternate explicitly specified destination.

No matter how you choose to migrate your data, however, 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 *GemStone/S 64 Bit 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 v3.0, 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`.

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 9.4

```
| 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 9.5 uses `migrateInstances:to:` to migrate all instances of all versions of a class, except the latest version, to the latest version.

Example 9.5

```
| animalHist allAnimals |
animalHist := Animal classHistory.
allAnimals := SystemRepository listInstances: animalHist.
"Returns an array of the same size as the class history.
 Each element in the array is a set corresponding to one
 version of the class.  Each set contains all the
 instances of that version of the class."

1 to: animalHist size-1 do: [:index | (animalHist at: index)
      migrateInstances:(allAnimals at: index)
      to: (animalHist at: animalHist size)].
```

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 202.

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 6, “Querying.”

For more information about committing and aborting transactions, see Chapter 7, “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 9.6.

Example 9.6

```
| 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 9.7 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 9.7

```
| 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 9.8.

Example 9.8

```
instVarMappingTo: anotherClass
| result myNames itsNames dietIndex |
"Use the default strategy first to properly fill in inst
vars having 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 9.9

```
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            4.7229757568719322E+02
  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 9.10

```
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

10 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 to create operating system sockets and exchange data between two independent interface processes.

10.1 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 10.1 creates a file named aFileName in the current directory on the client machine.

Example 10.1

```
| myFile myFilePath |
myFilePath := 'aFileName'.
myFile := GsFile openWrite: myFilePath.
"Here would go code to write data to the file"
myFile close
%
```

Example 10.2 creates a file named aFileName in the current directory on the server.

Example 10.2

```
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 217.

Opening and Closing a File

GsFile provides a wide variety of protocol to open and close files. For a complete list, see the image.

These methods return the GsFile instance if successful, or nil if an error occurs

Table 10.1 GsFile Method Summary

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.
`GsFile closeAll`	Closes all open GsFile instances on the client machine except stdin, stdout, and stderr.
`GsFile 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 10.3

```
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.

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 10.3) returns the entire contents of the specified file and positions the pointer at the end of the file.

In Example 10.4, `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 10.4

```
| 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. For example, the following code allows you to view the next character in the file without advancing the pointer.

Example 10.5

```
myFile peek
```

Example 10.6 allows you to advance the pointer by 16 characters without reading the intervening characters.

Example 10.6

```
myFile skip: 16
```

Testing Files

The class GsFile provides a variety of methods that allow you to determine facts about a file. For example, the following code tests to see whether the specified file exists on the client machine:

Example 10.7

```
GsFile exists: '/tmp/myfile.txt'
```

This method returns true if the file exists, false if it does not, and nil if an error occurred. To determine if the file exists on the server machine, use the method `existsOnServer:` instead.

Renaming Files

Files on the client or server can be renamed or moved. For example:

Example 10.8

```
GsFile rename: '/tmp/myfile.txt' to: '/tmp/newname.txt'.
```

Example 10.9

```
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:

Example 10.10

```
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 10.11

```
GsFile contentsOfDirectory: '$GEMSTONE/examples/admin'
onClient: false
%
an Array
  #1 /dbf/gsadmin/GS6430/examples/admin/.
  #2 /dbf/gsadmin/GS6430/examples/admin/..
  #3 /dbf/gsadmin/GS6430/examples/admin/onlinebackup.sh
  #4 /dbf/gsadmin/GS6430/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 10.11. 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 *. Example 10.12 shows a different use of this message.

Example 10.12

```
GsFile contentsOfDirectory: '$GEMSTONE/ver*' onClient: false
%
an Array
  #1 /dbf/gsadmin/GS6430/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 10.13

```
| 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 10.14

```
| myFile |
myFile := GsFile openRead: 'privatefile'.
myFile isNil
  ifTrue: [GsFile lastErrorString]
  ifFalse: ['Succesfully opened'].
%
errno=13,EACCES, Authorization failure (permission denied)
```

10.2 Executing Operating System Commands

System also understands the message `performOnServer:` *aString*, which causes the UNIX shell commands given in *aString* to execute in a subprocess of the current GemStone process. The output of the commands is returned as a GemStone Smalltalk string. For example:

Example 10.15

```
System performOnServer: 'date'
%
Mon Mar 21 15:19:56 PDT 2011
```

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:`.

10.3 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.

File in and file out can be done via ClassOrganizer. See Example 10.16.

Example 10.16

```
| myFile |
myFile := GsFile openWrite: 'UserGlobalsFileout.gs'.
myFile isNil
  ifTrue: [^GsFile serverErrorString].
ClassOrganizer new fileOutClassesAndMethodsInDictionary:
      UserGlobals on: myFile.
myFile close.
%
```

10.4 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 10.17.

Example 10.17

```
| 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 10.18

```
| empFile passivatedEmployees |
empFile := GsFile openReadOnServer: 'allEmployees.obj'.
passivatedEmployees := PassiveObject newOnStream: empFile.
AllEmployees := passivatedEmployees activate.
empFile close.
```

Examples 10.17 and 10.18 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.

10.5 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.

The class GsSocket includes two class methods, `clientExample` and `serverExample`, that provide an example of how you can create a GsSocket 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 in a GemBuilder for Smalltalk workspace.

The examples create a socket, establish a connection between them, exchange data using instances of PassiveObject, and then close the socket.

NOTE

The method `serverExample` *will take control of the interface that invokes it, allowing no further user input until the socket it creates succeeds in connecting to the client socket. If this happens, you need to interrupt the command.*

To run this example, execute the expression `GsSocket serverExample` from one interface before invoking the expression `GsSocket clientExample` from the other interface.

Chapter
11 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.

11.1 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.

11.2 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 234.

Example 11.1 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 11.1

```
"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 11.2 shows the notify set containing both the collection object and the elements in the collection.

Example 11.2

```
"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 — for a single session or across all sessions — without significant impact. Beyond that, test whether the response times are acceptable for your application.

If performance is a problem, you can set up a more formal system of change recording:

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 th 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
```

NOTE

To avoid missing intermediate changes to objects in your notify set, do

not clear your notify set after each transaction and then add some of the same objects to it again.

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 11.3 shows one way to do steps 5 and 6.

Example 11.3

```
"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 234.

Polling for Changes to Objects

You also use `System class>>signaledObjects` to poll for changes to objects in your notify set.

Example 11.4 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 11.4

```
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. Example 11.5 shows several common operations that trigger notification on the root object.

Example 11.5

```
"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 11.6 shows a method that creates an object and automatically adds it to the notify set in the process.

Example 11.6

```
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 12, "Handling Exceptions", on page 245, for more on handling Exceptions such as ObjectsCommittedNotification.

11.3 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

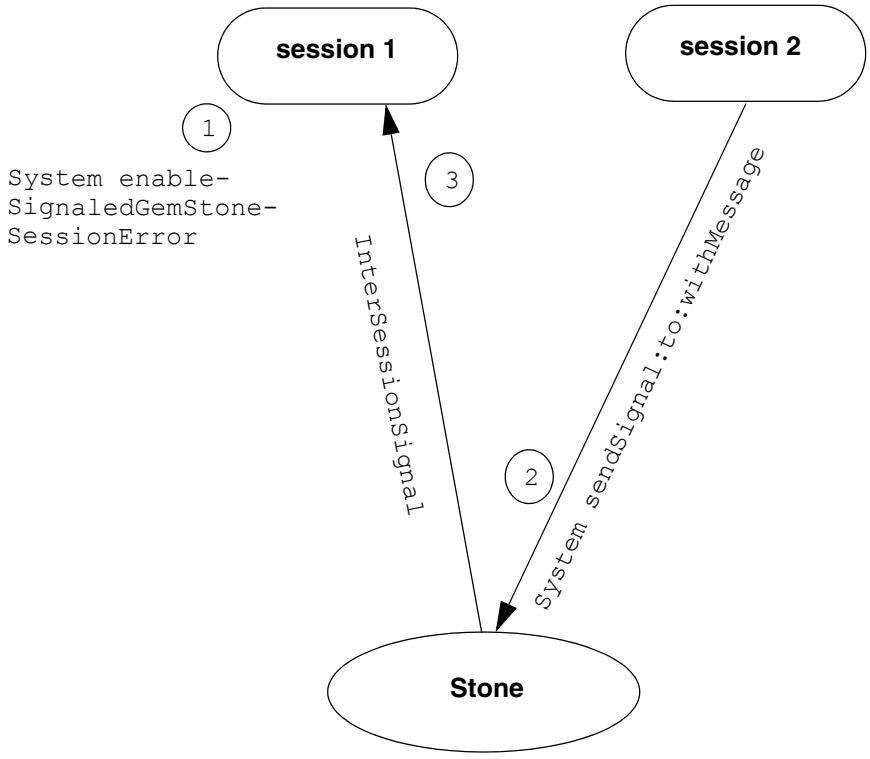
A signal is not an interrupt, and it does not automatically awaken an idle session. The signal can be received only when your session is actively executing Smalltalk code.

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 11.1 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 241.)
3. The Stone sends the exception `InterSessionSignal` to session1. The receiving session processes the signal with an exception handler. For details, see Chapter 12, Handling Exceptions.

Figure 11.1 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 11.7 shows how you might use this method to find the session ID for user1 and send a message.

Example 11.7

```
| 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 11.7 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 11.7 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 11.8.

Example 11.8

```
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 11.9.

For more complex information, the message could be a code where each symbol conveys its own meaning.

You can use signals to broadcast a message to every user logged in to GemStone. In Example 11.9, one session notifies all current sessions that it has created a new object to represent a stock that was added to the portfolio. In applications that commit whenever a new object is created, this code could be part of the instance creation method for class Holding. Otherwise, it could be application-level code, triggered by a commit.

Example 11.9

```
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 11.10 shows how to poll for Gem-to-Gem signals. If the polling process finds a signal, it immediately checks for another one until the queue is empty. Then the process sleeps for 10 seconds.

Example 11.10

```
| 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.

11.4 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.

Increasing Speed

No matter how you set up your application, signals and notifiers require a few milliseconds to get to their destination. You can improve the speed by using linked Gems, rather than separate RPC sessions.

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 230.

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 10, "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 11.11 shows some of the code executed by an error handler installed in a monitor Gem. It traps Gem-to-Gem signals and writes them to a log file.

Example 11.11

```
| 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

12 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 an unusual or undesired 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.

12.1 The Exception Class Hierarchy

GemStone/S 64 Bit v3.0 provides native support for ANSI Exceptions. The ANSI Exception framework defines subclasses to match the granularity of errors that you may want to handle.

In order to define, create, or signal exceptions, you must use the ANSI protocol. If you have defined custom exceptions based on the legacy interface, you will need to redefine these in version 3.0.

Figure 12.1 shows the ANSI exception handler class hierarchy. Note the following:

- Exception is the highest class that your Smalltalk code should ever handle.
- TestFailure and its subclass are used by the SUnit testing framework. For details about SUnit, see Chapter 17, beginning on page 323.

Figure 12.1 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  ( errorDescription externalSession )
              GciCompileError  ( list )
              GciEventError
              GciPause
              GciRuntimeError
              GciApplicationError
              GciDoesNotUnderstand
           ImproperOperation  ( object )
              ArgumentError
              ArgumentTypeError  ( expectedClass actualArg )
              CannotReturn
              LookupError  ( key )
              OffsetError  ( maximum actual )
              OutOfRange  ( minimum maximum actual )
                    FloatingPointError
              RegexpError
           IndexingErrorPreventingCommit
           InternalError
              GciTransportError
           InvalidUtf8Error
           LockError  ( object )
           MarkerNotFound
           NameError  ( selector )
              MessageNotUnderstood  ( envId receiver )
           NumericError
              ZeroDivide  ( dividend )
           RepositoryError
           SecurityError
           SignalBufferFull
           ThreadError
           TransactionError
```

```
      UncontinuableError
      UserDefinedError
         ExceptionSample
         ExceptionSampleNotResumable
         ExceptionSampleResumable
   Notification
      Admonition
         AlmostOutOfMemory
         AlmostOutOfStack
         RepositoryViewLost
      Deprecated
      FloatingPointException
      InterSessionSignal  ( sendingSession signal )
      ObjectsCommittedNotification
      TransactionBacklog  ( inTransaction )
      Warning
         CompileWarning
   TestFailure
      ResumableTestFailure
```

12.2 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. (ANSI errors include MessageNotUnderstood and ZeroDivide.)

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 250), 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 255) 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 12.1

```
method: Employee
age: anInt
(anInt between: 15 and: 65)
      ifFalse: [Error signal: 'Employee age out of range'].
age := anInt.
%
```

12.3 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.

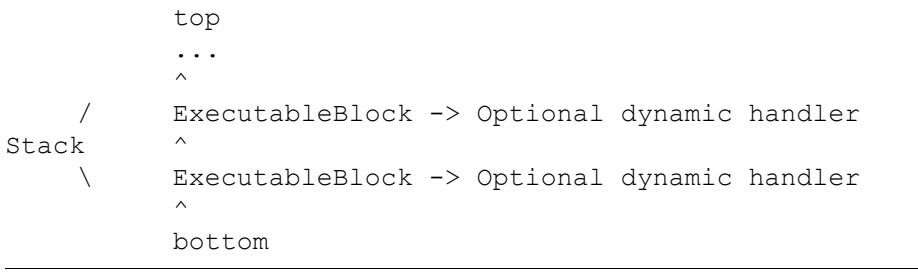
GemStone Smalltalk allows you to define two kinds of exception handlers: *dynamic (stack-based) handlers* and *default (static) handlers.*

Dynamic (Stack-Based) Handlers

A dynamic (stack-based) handler is associated with an ExecutableBlock 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 ExecutableBlock and applies as long as the ExecutableBlock is being executed. Because an ExecutableBlock can be embedded in another ExecutableBlock (either directly or via another method), multiple dynamic handlers can be active at one time. Figure 12.2 illustrates this relationship.

Figure 12.2 ExecutableBlock and Associated Handlers



To define a dynamic handler for an ExecutableBlock, send the `on:do:` message to the block. Example 12.2 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 12.2

```
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 ExecutableBlock 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 12.3 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 12.3

```
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 12.3, the handler for Error is installed first, and catches any Error or subclass exception signaled during the block that begins with `self doStuffA`.

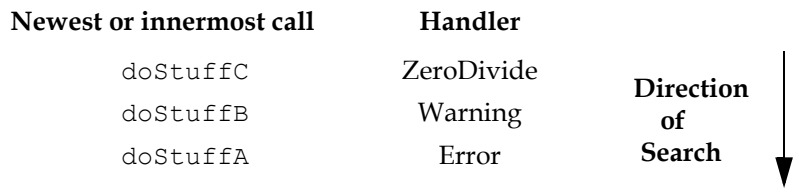
The handler for Warning is installed next, and catches any Warning or subclass exception signaled during the block that begins with `self doStuffB`.

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 12.3 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 ExecutableBlock 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 12.3 (on page 252), 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 12.3, 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 12.3, 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 12.3, 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 261). 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.

Chapter

13 The Legacy Exception Handling Framework

As discussed in Chapter 12, ANSI exception handling is the primary mechanism for dealing with errors in your programs. The legacy handler protocol has been deprecated with GemStone/S 64 Bit v3.0.

In v3.0, all exceptions are ANSI exceptions. Methods that previously signaled legacy exceptions now signal ANSI exceptions. For backwards compatibility, version 3.0 supports the use of legacy handlers to catch ANSI exceptions.

We strongly encourage the use of ANSI protocol for handling all errors. Nonetheless, you may continue to use the legacy protocol for handling GemStone system errors.

This chapter describes the legacy exception framework.

NOTE

In previous GemStone/S 64 Bit versions, the legacy exception framework included error categories. In version 3.0, categories are no longer supported. If you signaled and handled your own error categories in a previous GemStone version, you must rewrite your application to use ANSI exceptions in version 3.0.

Dynamic (Stack-Based) Exception Handlers
describes the use of dynamic legacy exception handlers, which are associated with a method being executed.

Default (Static) Exception Handlers
describes static exception handlers, which take control in the event of any error for which no other handler has been defined. A static handler executes without changing the stack, or the return value of the method that called it.

Flow of Control
discusses the flow of control in exception handlers.

Raising Exceptions
describes class methods on `System` that can be called to raise exceptions.

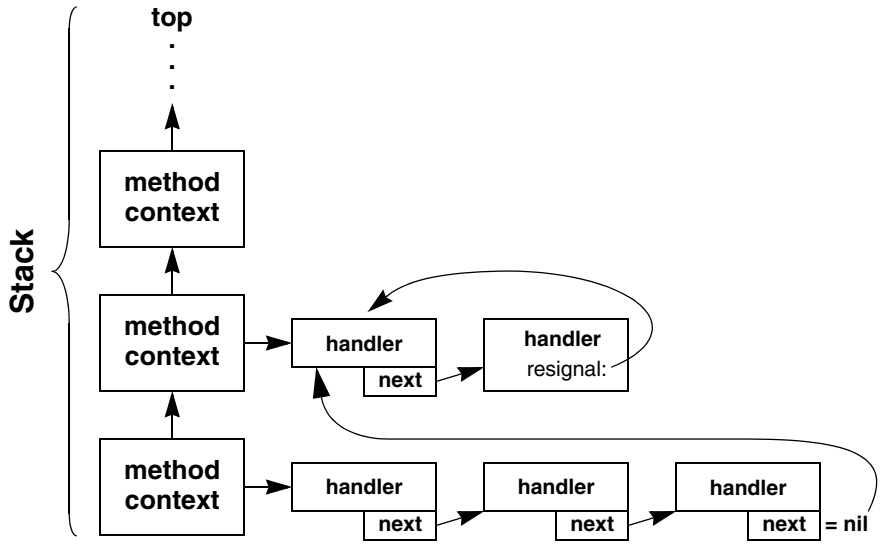
ANSI Integration
describes how to use legacy exception handlers to catch signaled ANSI exceptions.

13.1 Dynamic (Stack-Based) Exception Handlers

In ANSI, a dynamic (stack-based) exception handler is associated with an ExecutableBlock. 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 13.1.

Figure 13.1 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.

NOTE

In previous GemStone versions, the legacy exception framework included error categories. In GemStone/S 64 Bit v3.0, categories are no longer supported. If you

signaled and handled your own error categories in a previous GemStone version, you must rewrite your application to use ANSI exceptions in version 3.0.

- 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 13.1 catches the GemStone exception ZeroDivide and returns either PlusInfinity or MinusInfinity, depending on the sign of the dividend.

Example 13.1

```
| 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.

13.2 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 13.1. 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.

  NOTE

  In previous GemStone versions, the legacy exception framework included error categories. In GemStone/S 64 Bit v3.0, categories are no longer supported. If you signaled and handled your own error categories in a previous GemStone version, you must rewrite your application to use ANSI exceptions in version 3.0.

- 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 13.2

```
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 13.1 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 11, "Signals and Notifiers".

In Table 13.1, the legacy error symbol (and number) is listed along with the corresponding GemStone/S 64 Bit v3.0 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 12).

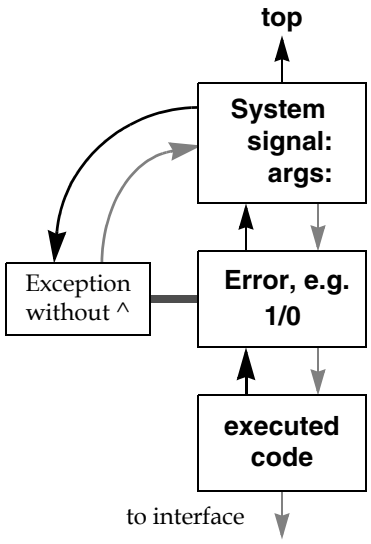
Table 13.1 Common GemStone Event Exceptions

Version 3.0 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 the method **`enableSignaledAbortError`**. No arguments. 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 the method **`enableSignaledFinishTransactionError`**.
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 the method **`enableSignaledObjectsError`**. No arguments.
InterSessionSignal #*rtErrSignalGemStoneSession (6010)*	Your session received a signal from another GemStone session. This error is received only if you have executed the method **`enableSignaledGemstoneSessionError`**. 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.
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 the method **`enableSignalTranlogsFull`** in this session.
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. No arguments.

13.3 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 13.1 on page 260. See Figure 13.2.

Figure 13.2 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 13.3 redefines how the GemStone exception #ZeroDivide is to be handled.

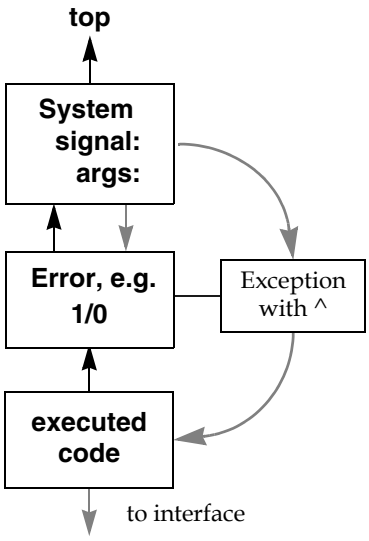
Example 13.3

```
| 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 13.3 shows the flow of control in Example 13.3.

Figure 13.3 Dynamic (Stack-Based) Exception Handler with Explicit Return



When you raise an error in a user action, you need to install an exception handler that explicitly returns, or the exception block may not leave the activation record stack in the correct state for continued execution. If the exception block does not contain an explicit return, the call to `userAction` should be placed by itself inside a method similar to this:

Example 13.4

```
callAction: aSymbol withArgs: args
^ System userAction: aSymbol withArgs: args
%
```

Signaling Other Exception Handlers

Under certain circumstances, your exception handler can choose to pass control to a previously defined exception handler, one that is below the present exception handler on the stack. To do so, your exception handler can send the message `resignal:number:args:`.

- The argument to the `resignal:` keyword is 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 13.5

```
| 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.

13.4 Raising Exceptions

NOTE

Legacy methods for raising exceptions continue to be available in GemStone/S 64 Bit v3.0 but now raise an ANSI exception, rather than a legacy exception.

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.

13.5 ANSI Integration

The ANSI and legacy frameworks should work together so that signaling an ANSI exception is caught by a legacy exception handler.

Example 13.6 shows a sample use of a legacy handler to catch signaled ANSI exceptions.

Example 13.6

```
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 Tuning Performance

GemStone Smalltalk includes several tools to help you tune your applications for faster performance.

Clustering Objects for Fast Retrieval
discusses how you can cluster objects that are often accessed together so that many of them can be found in the same disk access. Unnecessarily frequent disk access is ordinarily the worst culprit when application performance is not up to expectations.

Optimizing for Faster Execution
describes the profiling tool that allows you to pinpoint the problem areas in your application code and rewrite it to use more efficient mechanisms.

Modifying Cache Sizes for Better Performance
explains how to increase or decrease the size of various caches in order to minimize disk access and storage reclamation, both of which can significantly slow your application.

Managing VM Memory
discusses issues to consider when allocating and managing temporary object memory, and presents techniques for diagnosing and addressing OutOfMemory conditions.

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. The *GemStone/S 64 Bit System Administration Guide* provides a detailed discussion of the VSD utility.

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.

Cluster Buckets and Extents

You can determine the extentId of a given cluster bucket by executing an expression of the form:

```
aClusterBucket extentId
```

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 an unordered collection are created and modified using the cluster bucket associated with the specific collection, if any. To change the clustering of an unordered 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 Transactions

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".

A maximum of GEM_TEMPOBJ_CACHE_SIZE objects can be clustered in a single transaction.

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 5. 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 14.1 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). Returns true if the receiver was already clustered in the current transaction.
`clusterDescription`	Clusters in depth-first order those instance variables in the receiver that describe the structure of the receiver's instances. (Does not cluster the receiver itself.) The instance variables clustered are *instVarNames*, *classVars*, *categories*, and *class histories*.
`clusterBehaviorExceptMethods:` *aCollectionOfMethodNames*	This method can sometimes provide a better clustering of the receiving class and its method dictionary by omitting those methods that are seldom used. This omission allows frequently used methods to be packed more densely.

The code in Example 14.5 clusters class Employee's structure-describing variables, then its class methods, and finally its instance methods.

Example 14.5

```
| behaviorBucket descriptionBucket |
behaviorBucket := AllClusterBuckets at: 4.
descriptionBucket := AllClusterBuckets at: 5.
System clusterBucket: descriptionBucket.
Employee clusterDescription.
System clusterBucket: behaviorBucket.
Employee class clusterBehavior.
Employee clusterBehavior.
%
```

The next example clusters all of class Employee's instance methods except for `address` and `address:`

```
Employee clusterBehaviorExceptMethods: #(#address #address:).
```

Maintaining Clusters

Once you have clustered certain objects, they do not necessarily stay that way forever. You may therefore wish to check an object's location, especially if you suspect that such declustering is causing your application to run more slowly than it used to.

Determining an Object's Location

To enable you to check your clustering methods for correctness, Class Object defines the message `page`, which returns an integer identifying the disk page on which the receiver resides. For example:

```
anEmp page
2539
```

Disk page identifiers are returned only for temporary use in examining the results of your custom clustering methods—they are not stable pointers to storage locations. The page on which an object is stored can change for several reasons, 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 Optimizing for Faster 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.

The Class ProfMonitor

The class ProfMonitor allows you to sample the methods that are executed in a given block of code and to estimate the percentage of total execution time represented by each method. ProfMonitor starts a timer that determines which method is executing at specified intervals for a specified period of time. When it is done, it collects the results and returns them in the form of a string formatted as a report.

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:`.

You can specify the interval at which ProfMonitor checks to see which method is executing. By default, ProfMonitor checks execution every 1 ms. You can use ProfMonitor's instance method `interval:`, or a method with that keyword, to specify a shorter or longer interval in milliseconds.

When you need to profile operations that take very little time, you can specify the interval in nanoseconds (a nanosecond is a billionth of a second). The minimum interval is 1000 nanoseconds. To specify the interval in nanoseconds, use methods with the keyword `intervalNs:`.

It may be convenient to refer to Table 14.2 when determining the sample interval and reading the results:

Table 14.2 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, even once. It may be more useful to limit the reporting to methods that execute a certain number of times or more, so you see only the code that uses significant execution time. To do this, use ProfMonitor's instance method `reportDownTo:` *anInteger* or methods with the keyword `downTo:`.

Profiling Your Code

Profiling a Block of Code

To determine the execution profile for a piece of code, the simplest way is to format it as a block of code. This block needs to take long enough to execute that there will be a reasonable number of samples taken during execution. It can then be provided as the argument to `monitorBlock:`. Example 14.6 uses the default interval of 1 ms and includes every method it finds in its results, even those executing only once.

Example 14.6

```
ProfMonitor monitorBlock: [ 100 timesRepeat:
      [ System myUserProfile dictionaryNames ] ]
```

Example 14.7 uses a variant of `monitorBlock:` to check every 5 ms and include only methods that were found executing more than once.

Example 14.7

```
ProfMonitor
      monitorBlock: [ 100 timesRepeat:
            [ System myUserProfile dictionaryNames ]]
      downTo: 2
      interval: 5
```

In Example 14.8, the sampling is done every 5000 ns (5 us), so a much more detailed profile report will be produced for the same code block.

Example 14.8

```
ProfMonitor
      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 use the instance methods `startMonitoring` and `stopMonitoring`. The instance method `gatherResults` tallies the methods that were found, and the instance method `report` returns a string formatting the results.

Example 14.9 creates a new instance of ProfMonitor and changes the default file it will use to tally the results. It then starts profiling, executes the code to be profiled, stops profiling, tallies the methods encountered, and reports the results.

Example 14.9

```
| aMonitor |
aMonitor := ProfMonitor newWithFile: 'profile.tmp'.
aMonitor startMonitoring.
100 timesRepeat: [ System myUserProfile dictionaryNames ].
aMonitor stopMonitoring; gatherResults; report.
```

The class method `profileOn` also initiates profiling. You can use it, for example, to profile code contained in a file-in script. Example 14.10 shows how to do this using Topaz format.

Example 14.10

```
! get a profile report on a filein script
run
UserGlobals at: #Monitor put: ProfMonitor profileOn
%
input testFileScript.gs
! turn off profiling and get a report
run
Monitor profileOff
%
```

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.

The Profile Report

The profiling methods discussed in Examples 14.6 through 14.10 return a string formatted as a report. Example 14.11 shows a sample run.

Example 14.11

```
topaz 1> printit
ProfMonitor
     monitorBlock:[
            200 timesRepeat:[ System myUserProfile dictionaryNames] ]
     downTo: 2
%

================
STATISTICAL SAMPLING RESULTS
elapsed CPU time:    20 ms
monitoring interval: 1.0 ms

 tally       %   class and method name
------   -----   -------------------------------------
     7   30.43   IdentityDictionary        >> associationsDo:
     5   21.74   AbstractDictionary        >> _at:
     3   13.04   block in AbstractDictionary >> associationsDetect:ifNone:
     3   13.04   Array                     >> _at:
     2     8.7   Object                    >> _basicSize
     3   13.04   11 other methods
    23  100.00   Total


================
STATISTICAL STACK SAMPLING RESULTS
elapsed CPU time:    20 ms
monitoring interval: 1.0 ms

 total       %   class and method name
------   -----   -------------------------------------
    22   95.65   SymbolList                >> namesReport
    22   95.65   UserProfile               >> dictionaryNames
    21    91.3   SymbolList                >> names
    20   86.96   IdentityDictionary        >> associationsDo:
    20   86.96   AbstractDictionary        >> associationsDetect:ifNone:
     9   39.13   block in executed code
     5   21.74   AbstractDictionary        >> _at:
     3   13.04   block in AbstractDictionary >> associationsDetect:ifNone:
     3   13.04   Array                     >> _at:
     2     8.7   ProfMonitor               >> monitorBlock:
     2     8.7   Object                    >> _basicSize
     3   13.04   5 other methods
    23  100.00   Total
```

```
================
STATISTICAL METHOD SENDERS RESULTS
elapsed CPU time:    20 ms
monitoring interval: 1.0 ms

     %       %                     Parent
  self  total   total  local  Method
  Time   Time      ms    %         Child
------ ------  ------  -----  -----------

                   19.1    100       UserProfile  >> dictionaryNames
=    0.0   95.7    19.1    0.0  SymbolList   >> namesReport
                   18.3   95.5       SymbolList   >> names
                    0.9    4.5       String       >> add:
-------------------------------------------------------

                    7.8   40.9       block in executed code
=    0.0   95.7    19.1    0.0  UserProfile  >> dictionaryNames
                   19.1    100       SymbolList   >> namesReport
-------------------------------------------------------

                   18.3    100       SymbolList   >> namesReport
=    0.0   91.3    18.3    0.0  SymbolList   >> names
                   17.4   95.2       AbstractDictionary >>
associationsDetect:ifNone:
                    0.9    4.8       Array        >> add:
-------------------------------------------------------

                   17.4    100       AbstractDictionary >>
associationsDetect:ifNone:
=   30.4     87    17.4     35  IdentityDictionary >> associationsDo:
                    1.7     10       Object       >> _basicSize
                    2.6     15       block in AbstractDictionary >>
associationsDetect:ifNone:
                    2.6     15       Array        >> _at:
                    4.3     25       AbstractDictionary >> _at:
-------------------------------------------------------

                   17.4    100       SymbolList   >> names
=    0.0     87    17.4    0.0  AbstractDictionary >>
associationsDetect:ifNone:
                   17.4    100       IdentityDictionary >> associationsDo:
-------------------------------------------------------
```

```
                    0.9   11.1       ProfMonitor  >> monitorBlock:
=    0.0   39.1     7.8    0.0  block in executed code
                    7.8    100       UserProfile  >> dictionaryNames
---------------------------------------------------

                    4.3    100       IdentityDictionary >> associationsDo:
=   21.7   21.7     4.3    100  AbstractDictionary >> _at:
---------------------------------------------------

                    2.6    100       IdentityDictionary >> associationsDo:
=     13     13     2.6    100  block in AbstractDictionary >>
associationsDetect:ifNone:
---------------------------------------------------

                    2.6    100       IdentityDictionary >> associationsDo:
=     13     13     2.6    100  Array        >> _at:
---------------------------------------------------

                    0.9     50       ProfMonitor class >>
monitorBlock:downTo:
=    0.0    8.7     1.7    0.0  ProfMonitor  >> monitorBlock:
                    0.9     50       ProfMonitor  >> startMonitoring
                    0.9     50       block in executed code
---------------------------------------------------

                    1.7    100       IdentityDictionary >> associationsDo:
=    8.7    8.7     1.7    100  Object       >> _basicSize
---------------------------------------------------
```

As you can see, the report lists the methods that the profile monitor encountered when it checked the execution stack every millisecond. It sorts the methods according to the number of times they were found, with the most-often-used methods first. The ProfMonitor also calculates the percentage of total execution time represented by each method—useful information if you need to know where optimizing can do you the most good.

ProfMonitorTree

ProfMonitorTree performs the identical operations as does ProfMonitor, but produces an additional section in the final report, listing the profile results in tree form.

ProfMonitorTree understands the same protocol as ProfMonitor.

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 346 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

- Avoid concatenating strings. Instead, use the `add:` method to dynamically resize strings. This is much more efficient.

- If you have a choice between a method that modifies an object and one that returns a modified copy, use the method that modifies the object directly if your application allows it. This creates fewer temporary objects whose storage will have to be reclaimed. For example, `String >> ,` creates a new string to use in modifying the old one, whereas `String >> add:` modifies a string.

- Avoid generating temporary objects whose storage will need to be reclaimed. Storage reclamation can slow your application significantly.

- 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.

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 *GemStone/S 64 Bit 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 *GemStone/S 64 Bit 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 *GemStone/S 64 Bit 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 *GemStone/S 64 Bit 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 *GemStone/S 64 Bit 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 *GemStone/S 64 Bit 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 *GemStone/S 64 Bit 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 *GemStone/S 64 Bit 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 14.12

```
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 14.3 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 14.4 Statistics Related to Mark/Sweeps and Scavenges

NumberOfMarkSweeps	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 14.5 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 14.5 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.
MeSpaceAllocatedBytes	The number of bytes allocated for the remembered set (remSet), in-memory oopMap, and map entries.

Table 14.6 Statistics Related to Stubbing

NumRefsStubbedMarkSweep	The number of in-memory references that were stubbed (converted to a POM objectId) by in-memory mark/sweep.
NumRefsStubbedScavenge	The number of in-memory references that were stubbed (converted to a POM objectId) by in-memory scavenge.

Table 14.7 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 14.8 Statistics Related to Symbol Creation

NewSymbolRequests	The number of symbol creation requests by a session to the symbol creation gem.

Table 14.8 Statistics Related to Symbol Creation

NewSymbolsCount	The number of symbol creation requests by a session that did not resolve to an already committed symbol.
TimeWaitingForSymbols	Cumulative elapsed time (in milliseconds) waiting for symbol creation requests to be processed.

Table 14.9 Other Statistics

ExportedSetSize	The number of objects in the ExportSet (see page 296).
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 298), 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.

Objects that are intended to be persistent, but not log changes in the transaction logs, should be referenced by the variable NotTranloggedGlobals in the Globals SymbolDictionary.

Objects in NotTranloggedGlobals must have no other references from persistent objects. Objects that are reachable from AllUsers (the regular root for all persistent objects) may not reference anything that was previously only reachable from NotTranloggedGlobals; this 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. 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.

For more on transaction logs and backups refer to the *GemStone/S 64 Bit System Administration Guide*.

Chapter

15 Advanced Class Protocol

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.

You're already familiar with Object's protocol that enables an object to represent itself as a string. The class named Class defines the familiar subclass creation message (`subclass:instVarNames:...`) and some protocol for adding and removing class variables and pool dictionaries. Class Behavior defines by far the largest number of useful messages that you may not yet have encountered. This chapter presents a brief overview of Behavior's methods.

Adding and Removing Methods
describes the protocol in class Behavior for adding and removing methods.

Examining a Class's Method Dictionary
describes the protocol in class Behavior for examining the method dictionary of a class.

Examining, Adding, and Removing Categories
describes the protocol in class Behavior for examining, adding, and removing method categories.

Accessing Variable Names and Pool Dictionaries
describes the protocol in class Behavior for accessing the variables and pool dictionaries of a class.

15.1 Adding and Removing Methods

Class Behavior defines messages for adding or removing selectors.

Defining Simple Accessing and Updating Methods

Class Behavior provides an easy way to define simple methods for establishing and returning the values of instance variables. For each instance variable named by a symbol in the argument array, the message `compileAccessingMethodsFor:` *arrayOfSymbols* creates one method that sets the instance variable's value and one method that returns it. Each method is named for the instance variable to which it provides access.

For example, this invocation of the method:

```
Animal compileAccessingMethodsFor: #(#name)
```

has the same effect as the Topaz script in Example 15.1.

Example 15.1

```
category: 'Accessing'
method: Animal
name
   ^name
%
category: 'Updating'
method: Animal
name: aName
    name := aName
%
```

All of the methods created in this way are added to the categories named "Accessing" (return the instance variable's value) and "Updating" (set its value).

You can also use `compileAccessingMethodsFor:` to define methods for accessing pool variables and class variables. The only important difference is that to define *class* methods for getting at class variables, you must send

`compileAccessingMethodsFor:` to the *class* of the class that defines the class variables of interest. The following code, for example, defines class methods that access the class variables of the class Menagerie:

```
Menagerie class compileAccessingMethodsFor: #( #BandicootTally)
```

This is equivalent to the Topaz script in Example 15.2.

Example 15.2

```
category: 'Accessing'
classmethod: Menagerie
BandicootTally

    ^BandicootTally
%
category: 'Updating'
classmethod: Menagerie
BandicootTally: aNumber

    BandicootTally := aNumber
%
```

Removing Selectors

You can send `removeSelector:` *aSelectorSymbol* to remove any selector and associated method that a class defines. The following example removes the selector `habitat` and the associated method from the class Animal's method dictionary.

```
Animal removeSelector: #habitat
```

To remove a *class* method, send `removeSelector:` to the *class* of the class defining the method. The following example removes one of the class Animal's class methods:

```
Animal class removeSelector: #newWithName:favoriteFood:habitat:
```

The Basic Compiler Interface

Class Behavior defines the basic method for compiling a new method for a class and adding the method to the class's method dictionary. The programming environments available with GemStone Smalltalk provide much more convenient

facilities for most compilation jobs, so you may never need to use this method. It's useful for programmers who want to build a custom programming environment, or those who need to generate GemStone Smalltalk methods automatically.

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 many methods within Smalltalk will default to an environmentId of 0.

The following code compiles a short method named `habitat` for the class Animal, adding it to the category "Accessing":

Example 15.3

```
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 as shown in Example 15.3.

If `compileMethod:..` executes successfully, it adds the new method to the receiver and returns nil.

If the source string contains errors, this method returns an array of three-element arrays. Each three-element array includes an error number, an integer offset into

the source code string pointing to where the error was detected, and a string that describes the error.

15.2 Examining a Class's Method Dictionary

Class Behavior defines messages that let you obtain a class's selectors and methods, and determine if instance of a class can understand specific messages. You can get the source code or the compiled method itself.

For full protocol, see the image methods for Behavior under the category "Accessing the Method Dictionary". Some useful methods are:

```
Behavior >> allSelectors
Behavior >> canUnderstand: aSelector
Behavior >> compiledMethodAt: aSelector
Behavior >> includesSelector: aSelector
Behavior >> selectors
Behavior >> sourceCodeAt: aSelector
Behavior >> whichClassIncludesSelector: aSelector
```

Example 15.4 uses `selectors` and `sourceCodeAt:` in a method that produces a listing of the receiver's methods.

Example 15.4

```
classmethod: SomeClass
listMethods
"Returns a string listing the source code for the receiver's class
and instance methods."
| selectorArray outputString |
outputString := String new.
"First, concatenate all instance methods defined by the receiver."
outputString add: '************ Instance Methods *************';
            lf;
            lf.
selectorArray := self selectors.
selectorArray do: [:i | outputString
                      add: (self sourceCodeAt: i);lf;lf].
"Now add the class methods."
outputString add: '************* Class Methods *************';
            lf;lf.
selectorArray := self class selectors.
selectorArray do: [:i | outputString
                       add: (self class sourceCodeAt: i);lf;lf].
^outputString
%
```

Suppose that you defined a subclass of SymbolDictionary called CustomSymbolDictionary and used instances of that class in your symbol list for storing objects used in your programs.

The method in Example 15.5, using `includesSelector:`, would be able to tell you which of the classes in a CustomSymbolDictionary implemented a particular selector.

Example 15.5

```
printit
SymbolDictionary subclass: #CustomSymbolDictionary
  instVarNames: #()
  classVars: #()
  classInstVars: #()
  poolDictionaries: #()
  inDictionary: UserGlobals
%

method: CustomSymbolDictionary
whichClassesImplement: aSelector
  "Returns a string describing which classes in the receiver
  (a subclass of SymbolDictionary) implement a method whose selector
  is aSelector. Distinguishes between class and instance methods."

  | outputString |
  outputString := String new.
  self values do: [:aValue |
     (aValue isKindOf: Class) ifTrue: [
        (aValue includesSelector: aSelector) ifTrue: [
           outputString
              add: aValue name;
              add: ' (as instance method)';
              lf.
           ].
        (aValue class includesSelector: aSelector) ifTrue: [
           outputString
              add: aValue name;
              add: ' (as class method)';
              lf.
           ].
        ].
     ].
^outputString
%

printit
| newDict |
newDict := CustomSymbolDictionary new.
newDict at: #Dictionary put: Dictionary;
```

```
   at: #SymbolDictionary put: SymbolDictionary;
   at: #LanguageDictionary put: LanguageDictionary.
UserGlobals at: #myCustDict put: newDict.
System commitTransaction
%

printit
myCustDict whichClassesImplement: 'add:'
%
Dictionary (as instance method)
```

15.3 Examining, Adding, and Removing Categories

Class Behavior provides a nice set of tools for dealing with categories and for examining and organizing methods in terms of categories.

For full protocol, see the image methods for Behavior under the category "Accessing Categories". Some useful methods are:

```
Behavior >> addCategory: categoryName
Behavior >> categoryNames
Behavior >> moveMethod: aSelector toCategory: categoryName
Behavior >> removeCategory: categoryName
Behavior >> renameCategory: categoryName to: newCategoryName
Behavior >> selectorsIn: categoryName
```

Note that `removeCategory:` removes not only a category but all of the methods in that category.

Example 15.6 shows how you might move methods to another category before deleting their original category.

Example 15.6

```
Animal addCategory: 'Munging'.
(Animal selectorsIn: 'Accessing') do: [:i |
          Animal moveMethod: i toCategory: 'Munging'].
Animal removeCategory: 'Accessing'.
```

The next example demonstrates how you use these methods to list instance methods in a format that can be read and compiled automatically by Topaz with the `input` function. This partially duplicates the Topaz file-out function.

Example 15.7

```
classmethod: SomeClass
listMethodsByCategory
  "Produces a string describing the receiver's instance methods
  by category in FILEOUT format."

  | outputString newline className |
  outputString := String new.
  className := self name.
  self categoryNames do: "For each category..."
     [:aCatName |
     outputString
        add: 'category: ''';
        add: aCatName;
        add: '''';
        lf.
     (self selectorsIn: aCatName) do: "For each selector..."
        [:aSelector |
        outputString
           add: 'method: ';
           add:className; lf;
           add: (self sourceCodeAt: aSelector);
           add: newline;
           add: '%'; lf;lf.
           ].
     ].
^outputString
%
```

Here is how this method behaves if it were defined for class Animal. (The output is truncated.)

Example 15.8

```
printit
Animal listMethodsByCategory
%
category: 'Accessing'
method: Animal
name: aName

   name := aName

%

method: Animal
name

   ^name
%
```

15.4 Accessing Variable Names and Pool Dictionaries

Class Behavior's methods provide access to the names of all of a class's variables (instance, class, class instance, and pool).

The following methods return the class, instance, and pool variables defined by the receiver directly:

```
Behavior >> allClassVarNames
Behavior >> allInstVarNames
Behavior >> allSharedPools
```

These methods include both the variables defined by the receiver, and the variables inherited from superclassses:

```
Behavior >> classVarNames
Behavior >> instVarNames
Behavior >> sharedPools
```

For full protocol, see the image methods for Behavior under the category "Accessing Variables".

For example, the method in Example 15.9, defined for the class CustomSymbolDictionary (discussed earlier on page 309), returns a list of all classes named by the receiver that define some name as an instance, class, class instance, or pool variable.

Example 15.9

```
method: CustomSymbolDictionary
listClassesThatReference: aVarName
  "Lists the classes in the receiver, a subclass of
  SymbolDictionary, that refer to aVarName as an instance, class,
  class instance, or pool variable."

  | theSharedPools outputString |
  outputString := String new.
  self valuesDo: [:aValue |
     "For each value in the receiver's Associations..."
     (aValue instVarNames includesValue: aVarName) ifTrue: [
        outputString
           add: aValue name;
           add: ' defines ', aVarName, ' as inst variable.';
           lf.
        ].
     (aValue classVarNames includesValue: aVarName) ifTrue: [
        outputString
           add: aValue name;
           add: ' defines ', aVarName, ' as class variable.';
           lf.
        ].
     theSharedPools:= aValue sharedPools.
     theSharedPools do: [:poolDict |
        (poolDict includesKey: aVarName) ifTrue: [
           outputString
              add: aValue name;
              add: ' defines ', aVarName, ' as pool var'.
           ].
        ].
     ].
  ^outputString
%

printit
myCustDict listClassesThatReference: #tableSize
%
Dictionary defines tableSize as inst variable.
```

Chapter

16 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.

16.1 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*

C type symbols

Table 16.1 lists the symbols used for creating *resType* (result type) and *argumentTypes* arguments when creating CCallouts.

Table 16.1 C type symbols

	Return type	Argument type
#int64	Integer. The C function returns an int64, or any unsigned C integer smaller than 64 bits.	Integer
#uint64	Integer. The C function returns a uint64.	Integer
#int32	Integer. The C function returns a signed C integer 32 bits or smaller.	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	

Table 16.1 C type symbols (Continued)

	Return type	Argument type
#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.
#'&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.

Platform-specific limitations

Not all platforms support native code generation.

On platforms supporting native code:

- Functions using varArgs may have a maximum of 20 variable arguments.

On platforms that do not support native code:

- 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

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.

16.2 FFI Wrapper Utilities

In addition to the core classes described in the previous section, the FFI provides the class CHeader, along with three internal classes that are used by CHeader: CDeclaration, CPreprocessor, and CPreprocessorToken.

CHeader

You can use CHeader to generate wrappers for C functions and structures. The CHeader instance method

`wrapperNamed:` *nameString* **`forLibraryAt:`** *pathString* **`select:`** *aBlock*

generates a wrapper for the named library.

For example, the code in Example 16.1 generates a wrapper named ZLib and installs that wrapper class in UserGlobals. ZLib contains a wrapper for selected functions that are defined in the zlib compression library.

Example 16.1

```
| 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 |
      0 < (each name findString: 'flate' startingAt: 1) or: [
      0 < (each name findString: 'zlib' startingAt: 1) or: [
      0 < (each name findString: 'ompress' startingAt: 1) or: [
      0 < (each name findString: 'gz' startingAt: 1)
   ]]]].
class initializeFunctions.
UserGlobals at: class name put: class.
%
```

Example 16.2 demonstrates a trivial call to the ZLib library.

Example 16.2

```
ZLib new zlibVersion "answers the String '1.2.3.3'"
%
```

Example 16.3 demonstrates a more complex usage, involving zlib compression.

Example 16.3

```
| source destination size result |
source := 'Now is the time for all good men to come to the
aid of their party'.
destination := CByteArray gcMalloc: 100.
size := CByteArray gcMalloc: 8.
size int64At: 0 put: destination size.
result := ZLib new
      compress_: destination
      _: size
      _: source
      _: source size.
size := size int64At: 0.
destination := destination byteArrayFrom: 0 to: size - 1.
destination asArray.
   "anArray( 120, 156, 21, 138, 209, 9, 128, 48, 16, 197,
   86, 201, 68, 238, 80, 236, 85, 15, 90, 159, 156, 7, 226,
   246, 182, 95, 9, 36, 155, 94, 252, 33, 79, 35, 125, 24,
   77, 65, 233, 157, 67, 170, 12, 187, 72, 177, 107, 134,
   201, 53, 21, 175, 168, 45, 245, 224, 46, 145, 223, 15, 1,
   237, 23, 68)"
source size -> destination size.    "66->64"
%
```

Chapter
17 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.

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.

17.1 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.

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.

17.2 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.

17.3 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 17.1 defines the class ExampleSetTest, a subclass of TestCase.

Example 17.1

```
TestCase subclass: 'ExampleSetTest'
      instVarNames: #( full empty)
      classVars: #()
      classInstVars: #()
      poolDictionaries: #[]
      inDictionary: Globals
      instancesInvariant: false
      isModifiable: false
```

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 17.2 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 17.2

```
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 17.3 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 17.3

```
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 17.2, 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 17.4.

Example 17.4

```
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 17.5.

Example 17.5

```
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 17.6). 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 17.6

```
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 17.7

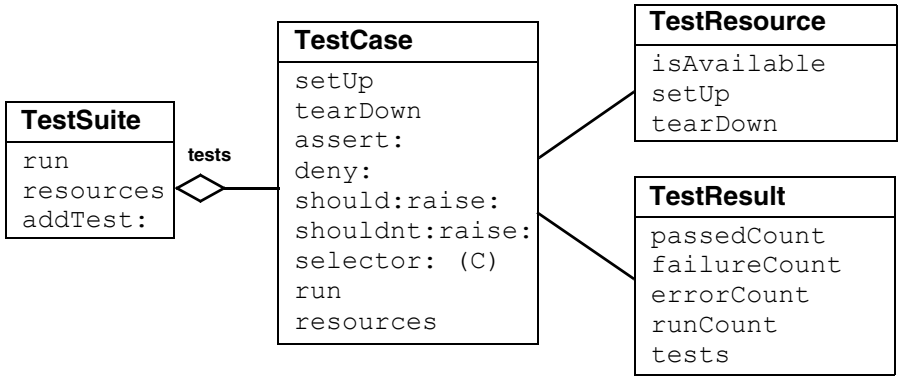
```
GSExampleSetTest>>testIllegal
   self should: [empty at: 5] raise: 2007.
   self should: [empty at: 5 put: #abc] raise: 2007.
```

Having provided an example of writing and running a test, we now turn to an investigation of the framework itself.

17.4 The SUnit Framework

SUnit is implemented by four main classes: TestSuite, TestCase, TestResult, and TestResource. See Figure 17.1. (Note that this is an object composition diagram, not a class hierarchy diagram.)

Figure 17.1 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 17.2 (on page 327), 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 17.8, 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 17.8

```
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.

17.5 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 17.2.

Figure 17.2 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 17.3) 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 17.3 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 17.4, the method `TestCase>>runCase` calls the methods `setUp` and `tearDown`.

Figure 17.4 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 17.5). 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 17.5 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 17.6.

Figure 17.6 TestResource class methods `isAvailable` and `current` (source code)

```
        TestResource class>>isAvailable
           ^self current notNil

        TestResource class>>current
           current isNil ifTrue: [current := self new].
           ^current
```

17.6 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.

Appendix

A GemStone Smalltalk Syntax

This chapter outlines the syntax for GemStone Smalltalk and introduces some important kinds of GemStone Smalltalk objects.

A.1 The Smalltalk Class Hierarchy

Every object is an instance of a class, taking its methods and its form of data storage from its class. Defining a class thus creates a kind of template for a whole family of objects that share the same structure and methods. Instances of a class are alike in form and in behavioral repertoire, but independent of one another in the values of the data they contain.

Classes are much like the data types (string, integer, etc.) provided by conventional languages; the most important difference is that classes define actions as well as storage structures. In other words, Algorithms + Data Structures = Classes.

Smalltalk provides a number of predefined classes that are specialized for storing and transforming different kinds of data. Instances of class Float, for example, store floating-point numbers, and class Float provides methods for doing floating-point arithmetic. Floats respond to messages such as +, -, and reciprocal.

Instances of class Array store sequences of objects and respond to messages that read and write array elements at specified indices.

The Smalltalk classes are organized in a treelike hierarchy, with classes providing the most general services nearer the root, and classes providing more specialized functions nearer the leaves of the tree. This organization takes advantage of the fact that a class's structure and methods are automatically conferred on any classes defined as its subclasses. A subclass is said to inherit the properties of its parent and its parent's ancestors.

How to Create a New Class

The following message expression makes a new subclass of class Object, the class at the top of the class hierarchy:

```
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.

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 GemStone 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 342). 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 341).

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 GemStone 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 GemStone Smalltalk, literal numbers look and act much like numbers in other programming languages. Like other GemStone Smalltalk objects, numbers receive and respond to messages. Most of those messages are requests for arithmetic operations. In general, GemStone Smalltalk numeric expressions do the same things as their counterparts in conventional programming languages. For example:

```
5 + 5
```

returns the sum of 5 and 5.

A literal floating point number must include at least one digit after the decimal point:

```
5.0
```

You can express very large and very small numbers compactly with scientific notation. To raise a number to some exponent, simply append the letter "e" and a numeric exponent to the number's digits. For example:

```
8.0e2
```

represents 800.0. The number after the `e` represents an exponent (base 10) to which the number preceding the `e` is to be raised. The result is always a floating point number. Here are more examples:

```
1e-3   represents 0.001
1.5e0  represents 1.5
```

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 literials
- "s" for ScaledDecimal literals
- "p" for FixedPoint literals

For details, see "GemStone Smalltalk Lexical Tokens" on page 365.

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 GemStone 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, GemStone 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 5, "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), GemStone Smalltalk's 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 351.

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 GemStone Smalltalk programmers begin local variable names with lowercase letters and global variable names with uppercase letters. When a variable name contains several words, GemStone Smalltalk programmers usually begin each word with an uppercase letter. You are free to ignore either of these conventions, but remember that GemStone Smalltalk is case-sensitive. The following are all different names to GemStone Smalltalk:

```
VariableName
variableName
variablename
```

Variable names can contain up to 1024 characters.

Declaring Temporary Variables

Like many other languages, 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 Smalltalk, is the temporary variable. Temporary variables are so called because they are defined only for one execution of the set of statements in which they are declared.

To declare a temporary variable, you must surround it with vertical bars as in this example:

Example A.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.

true

Refers to the object representing logical truth.

false

Refers to the object representing logical falsity.

self

Refers to the receiver of the message, which differs according to the context. self may be used anywere a method argument or method temporary would be used, except self is not allowed on the left side of an assignment.

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.

Assignment

Assignment statements in GemStone Smalltalk look like assignment statements in many other languages. The following statement assigns the value 2 to the variable MightySmallInteger:

```
MightySmallInteger := 2.
```

The next statement assigns the same String to two different variables (C programmers may notice the similarity to C assignment syntax):

```
nonmodularity := interdependence := 'No man is an island'.
```

Message Expressions

As you know, GemStone Smalltalk objects communicate with one another by means of messages. Most of your effort in GemStone 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

Because GemStone represents selectors internally as symbols, almost any identifier that is legal as a literal symbol is acceptable as a selector. A few selectors,

however, have been reserved for the sole use of the GemStone Smalltalk kernel classes. Those selectors are:

```
                    _isSmallInteger     ifTrue:ifFalse:
_isArray            _isSymbol           includesIdentical:
_isExceptionClass   ~~                  isKindOf:
_isExecBlock        ==                  timesRepeat:
_isFloat            ifFalse:            to:by:do:
_isInteger          ifFalse:ifTrue:     to:do:
_isNumber           ifNil:              untilFalse
_isOneByteString    ifNil:ifNotNil:     untilTrue
_isRange            ifNotNil:           whileFalse:
_isRegexp           ifNotNil:ifNil:     whileTrue:
_isRubyHash         ifTrue:
```

Redefining a reserved selector has no effect; the same primitive method is called and your redefinition is ignored.

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.2 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
2.0000000000000000E+05
```

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.3 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