GemStone

GemStone Kernel Reference

July 1996

GemStone

Version 5.0

IMPORTANT NOTICE

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

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

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

Trademarks

Ethernet is a registered trademark of Xerox.

GemStone is a registered trademark of GemStone Systems, Inc.

Kerberos is Copyright (C) 1989 by the Massachusetts Institute of Technology. Export of this software from the United States of America is assumed to require a specific license from the United States Government. It is the responsibility of any person or organization contemplating export to obtain such a license before exporting. Within that constraint, permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of M.I.T. not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. M.I.T. makes no representations about the suitability of this software for any purpose. It is provided “as is” without express or implied warranty.

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

Microsoft, **MS**, **MS-DOS**, **Windows**, **XENIX**, **CodeView**, and **QuickC** are registered trademarks and **QBasic**, **Windows NT**, and **Win32** are trademarks of Microsoft Corporation in the U.S.A. and other countries.

Preface

About This Manual

The *GemStone Kernel Reference* provides a reference for the GemStone® programming environment and detailed functional descriptions of the operations you can perform in it. It describes the objects of the GemStone kernel, including GemStone kernel classes.

Intended Audience

This manual is intended for programmers who are already familiar with the GemStone Smalltalk language and the GemStone object server. The material presented here is directed toward the reader who has specific questions about the workings of the elements of GemStone.

This manual assumes that the GemStone object server has been correctly installed on your host computer as described in the *GemStone System Administration Guide*, and that your system meets the requirements listed in your *GemStone Installation Guide*.

How This Manual is Organized

Chapter 1, Reserved and Predefined Objects
describes the GemStone Smalltalk class hierarchy and predefined kernel objects in the GemStone programming environment.

Chapter 2, Class Reference
describes each GemStone Smalltalk kernel class and the methods that are unique to that class.

Conventions

Class names are shown in regular typeface and begin with capital (uppercase) characters.

Instance variables are shown in **bold** typeface.

Messages and message selectors are shown in `monospace` typeface.

Message arguments are shown in *italic* typeface.

Other Useful Documents

This manual should be treated as the companion volume to the *GemStone Programming Guide*. That book describes the GemStone Smalltalk language and programming environment from an introductory and conceptual level, while this book serves as the corresponding reference.

Many topics mentioned in this volume pertain to GemStone systems administration, which are covered in detail in the *GemStone System Administration Guide*.

Technical Support

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

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

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

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

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

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

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

Email: **support@gemstone.com**

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

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

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

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

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

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

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

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

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

24x7 Emergency Technical Support

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

Contents

Chapter 1. Reserved and Predefined Objects

Chapter 2. Class Reference

Method Index

Index

Figures

Tables

Chapter 1

Reserved and Predefined Objects

This chapter outlines the relationships of the objects that comprise the GemStone kernel when GemStone is first installed. These objects consist primarily of classes, constants, collections such as dictionaries, and a few other supporting elements.

The root of these relationships is AllUsers, an instance of UserProfileSet, which contains the UserProfile objects of all GemStone users. Each user profile has a SymbolList object, which is an Array of SymbolDictionary objects. The entries in a user profile's symbol list prepare each session to resolve symbols at compile time.

Initially, each user profile has three dictionaries from the GemStone kernel in its symbol list: UserGlobals, Globals, and Published. The UserGlobals dictionary holds a few definitions by which a user might customize the environment. The Globals dictionary defines the GemStone kernel objects to which all GemStone users may need to refer. These objects comprise most of the GemStone kernel, including all of the GemStone kernel classes.

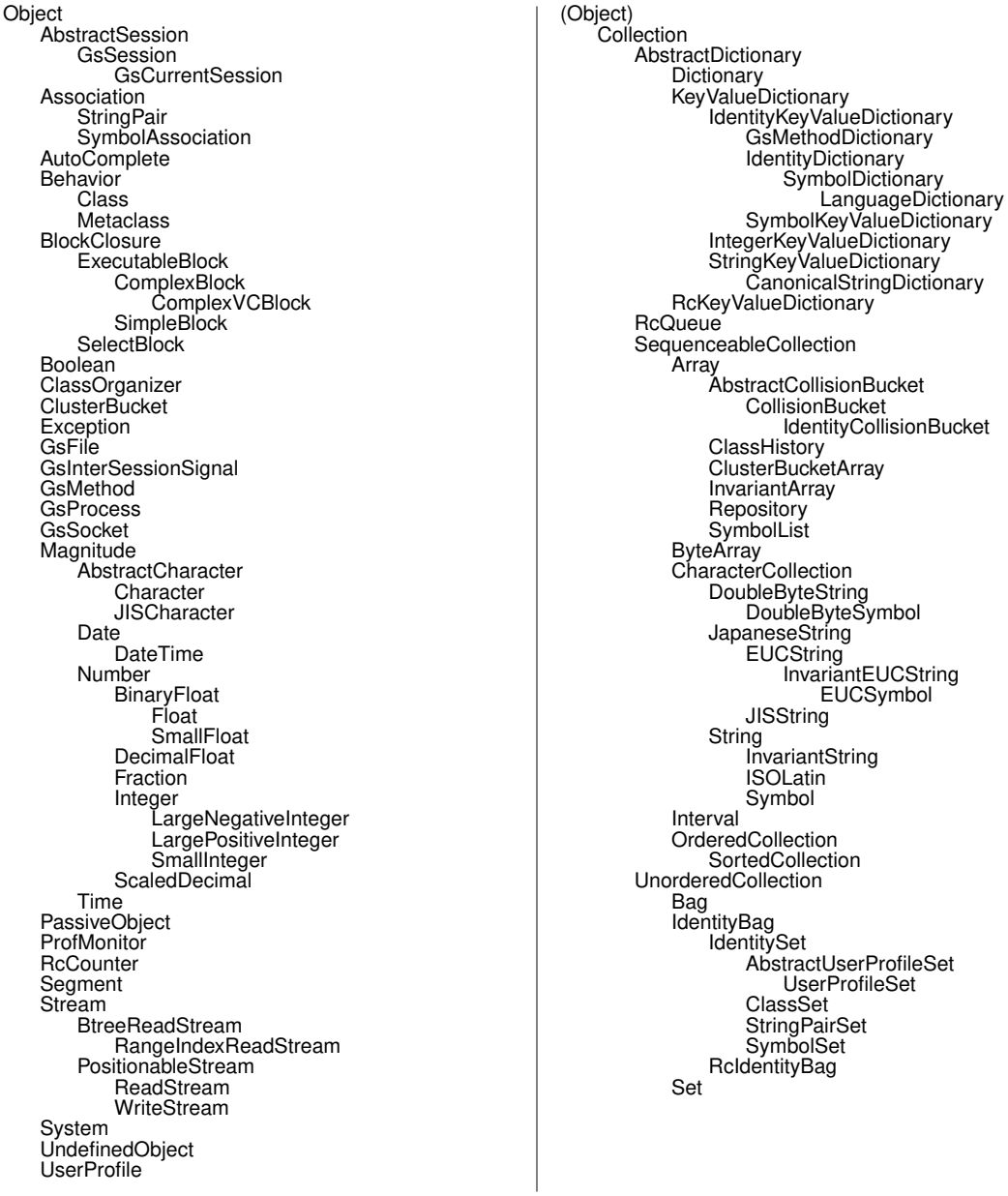
The Published Dictionary is empty in a fresh GemStone installation. A GemStone administrator can place objects to be shared by well-defined groups of users into the Published Dictionary.

1.1 The GemStone Smalltalk Class Hierarchy

In GemStone Smalltalk, classes inherit structure and behavior from other classes. A class's ancestry is as important as its own method definitions.

The GemStone Smalltalk class organization is hierarchical, with class Object at the top. Each object inherits fixed instance variables and methods from its class and from its class's superclasses. Figure 1.1 shows the GemStone kernel classes and how they fit into the GemStone Smalltalk class hierarchy.

Figure 1.1 The GemStone Smalltalk Class Hierarchy



1.2 Reserved Selectors and Optimized Selectors

The GemStone Smalltalk virtual machine optimizes and reserves the implementation of certain methods, as noted in their individual descriptions in Chapter 2. See the *GemStone Programming Guide* for a full list and discussion of the selectors for these methods.

1.3 Selectors for Private Methods

Each method whose selector begins with an underscore (_) character is reserved for use by the GemStone development team as a *private* method.

The methods described in this manual are *public* — that is, they are publicly documented here as part of the system protocol. Private methods are implementation-dependent and thus subject to change.

CAUTION:
Private methods are subject to change at any time. Do not depend on the presence or behavior of any private method when creating your own classes and methods.

1.4 GemStone Smalltalk Primitives

Some methods are implemented as compiler primitives in order to improve efficiency. The GemStone Smalltalk code for such methods begins with a statement of the form:

```
<primitive: 123>
```

Only SystemUser is permitted to compile a method using this syntax.

You can invoke primitives by means of *user actions*—functions callable from GemStone Smalltalk but written in other languages. For more information on user actions, refer to the *GemBuilder for C* manual.

1.5 Predefined GemStone Kernel Objects

This section describes the predefined objects that are located in a freshly installed GemStone database.

Users

The AllUsers object, an instance of UserProfileSet, contains the UserProfile objects of the users that are known to the database. Initially, it has three elements: **SystemUser**, **DataCurator**, and **GcUser**.

The **SystemUser** UserProfile is ordinarily used only for performing GemStone system upgrades. Certain system objects — including the GemStone-supplied kernel classes, along with their methods and class variables — are owned by the SystemUser and are stored in the System Segment. (That Segment also contains all instances of classes Character and SmallInteger.)

The **DataCurator** UserProfile is used to perform the data curator tasks described in the *GemStone System Administration Guide*. Most of the predefined GemStone objects listed in this section are owned by the DataCurator and are stored in the DataCurator Segment.

The **GcUser** UserProfile is used to control GemStone's garbage collection tasks. A person does not normally log into GemStone as **GcUser**. **GcUser** initially has only the GarbageCollection privilege. Its UserGlobals dictionary contains the parameters that control the reclaim and epoch garbage collection. See the *GemStone System Administration Guide* for more information about **GcUser**.

Dictionaries

UserGlobals. Each UserProfile object contains a symbol list, an Array of SymbolDictionaries that initialize a session so that it can resolve symbols at compile time. The first element in the symbol list is always the SymbolDictionary UserGlobals, which initially contains three keys:

- #UserGlobals (corresponding to the dictionary itself)
- #MinutesFromGmt (not used in this release)
- #NativeLanguage (initially #English)

Each user's UserGlobals dictionary is stored in that user's default Segment.

Globals. This SymbolDictionary defines the GemStone kernel classes and any other objects to which all GemStone users may need to refer. Figure 1.1 lists the contents of this directory when GemStone is first installed. The Globals dictionary is stored in the DataCurator Segment.

Published. This SymbolDictionary can be used to share objects among users. Objects in this dictionary can be read by one group (#Subscribers) and modified by another group (#Publishers). The #Publisher group can also be a subset of the #Subscriber group. The Published Dictionary is primarily an example, so it provides minimal object deployment capability. The Published dictionary is stored in the Published Segment.

Non-numeric Constants

true. This object (an instance of Boolean) is defined in the Globals dictionary, and is stored in the System Segment.

false. This object (an instance of Boolean) is defined in the Globals dictionary, and is stored in the System Segment.

nil. This object (an instance of UndefinedObject) is defined in the Globals dictionary and is stored in the System Segment.

Numeric Constants

Floating point constants are instances of class Float or class DecimalFloat. They are defined in the Globals dictionary and are stored in the System Segment. Refer to IEEE standards 754-1987 and 854-1987 for more information regarding their meanings in floating-point calculations.

DecimalPlusInfinity.

DecimalMinusInfinity.

DecimalPlusQuietNaN.

DecimalMinusQuietNaN.

DecimalPlusSignalingNaN.

DecimalMinusSignalingNaN.

PlusInfinity.

MinusInfinity.

PlusQuietNaN.

MinusQuietNaN.

PlusSignalingNaN.

MinusSignalingNaN.

Repository and Segments

SystemRepository. This Repository is defined in the Globals dictionary. The SystemRepository object itself is stored in the DataCurator Segment, and its indexable part contains references to all GemStone Segments.

GemStone represents all the disk space it uses as a single instance of class Repository. When GemStone is first installed, that Repository has the name SystemRepository. The SystemRepository object initially contains two segments: SystemSegment (owned by the SystemUser) and DataCuratorSegment (owned by the DataCurator). New Segments may be created (added to the SystemRepository object) when new users are added.

SystemSegment. This Segment is defined in the Globals dictionary, and is referenced from the indexable part of the SystemRepository. The SystemSegment object itself is stored in the DataCurator Segment.

The SystemSegment is the default Segment for its owner, the SystemUser (who has write authorization for any of the objects in this Segment). The "world" (the set of all GemStone users) is authorized to read, but not write, the objects in this Segment. In addition, the group #System is authorized to write in this Segment.

WARNING:

Logging in to GemStone as SystemUser is like logging in to your workstation as root — an accidental modification to a kernel object can cause a great deal of harm. Use the DataCurator account for all system administration operations except those that ***require*** *SystemUser privileges, such as upgrades and full restores.*

DataCuratorSegment. This Segment is defined in the Globals dictionary, and is referenced from the indexable part of the SystemRepository. The DataCuratorSegment object itself is stored in the DataCurator Segment.

The DataCuratorSegment is the default Segment for its owner, the DataCurator (who has write authorization for any of the objects in this Segment). The "world" (the set of all GemStone users) is authorized to read, but not write, the objects in this Segment. No groups are initially authorized to read or write in this Segment.

Objects in the DataCuratorSegment include the Globals dictionary, the SystemRepository object, all Segment objects, AllUsers (the set of all GemStone UserProfiles), AllGroups (the collection of groups authorized to read and write objects in GemStone segments), and each UserProfile object.

NOTE:

When GemStone is installed, only the DataCurator is authorized to write in this Segment. To protect the objects in the DataCurator Segment against unauthorized modification, other users should not write in this Segment.

PublishedSegment. This Segment is defined in the Globals dictionary, and is referenced from the indexable part of the SystemRepository. The PublishedSegment object itself is stored in the DataCurator Segment.

The PublishedSegment is owned by the SystemUser. The group #Subscribers is authorized to read in this Segment. The group #Publishers is authorized to read and write in this segment. The "world" is not authorized to read or write the objects in this Segment.

DbfHistory. DbfHistory is a String object residing in the DataCurator Segment that contains information regarding conversions and updates applied to the database.

NativeLanguage. This Symbol is defined in the Globals dictionary (with an initial value of #English), and may be redefined in each user's UserGlobals directory. The NativeLanguage object permits GemStone to return error messages and other interactive messages in any of the supported languages. (Initially, only English is supported.)

Global Collections

AllGroups. This CanonicalStringDictionary is defined in the Globals dictionary, and is stored in the DataCurator Segment. Each Symbol in AllGroups corresponds to a group of users who have been authorized to read or write in one or more Segments. When GemStone is first installed, AllGroups contains the single symbol #System.

AllSymbols. This CanonicalStringDictionary is defined in the Globals dictionary, and is stored in the DataCurator Segment. Initially, it contains references to all Symbols created with GemStone itself. AllSymbols is in the DataCurator segment and is readable by all users.

AllUsers. The AllUsers object (a UserProfileSet) is defined in the Globals dictionary, and is stored in the DataCurator Segment. AllUsers contains the UserProfiles of all GemStone users. When GemStone is first installed, AllUsers contains three UserProfiles: SystemUser, DataCurator, and GcUser.

AllClusterBuckets. This ClusterBucketArray is defined in the Globals dictionary, and is stored in the DataCurator Segment. Each Symbol in AllClusterBuckets corresponds to a cluster bucket, which organizes a physical storage specification for a group of objects. When GemStone is first installed, AllClusterBuckets contains seven symbols, for the following predefined cluster buckets (listed by cluster id):

1. A generic bucket whose extent is "don't care". This bucket, the current default after session login, is invariant and may not be modified. Making this bucket invariant increases the fault tolerance of the system, and facilitates both building the kernel database and database conversion.
2. A generic bucket whose extent is "don't care".
3. A generic bucket whose extent is "don't care".
4. The kernel classes "behaviorBucket", extent 1.
5. The kernel classes "descriptionBucket", extent 1.
6. The kernel classes "otherBucket", also used for AllSymbols, extent 1.
7. A generic bucket whose extent is "don't care".

ConfigurationParameterDict. This SymbolKeyValueDictionary is defined in the Globals dictionary, and is stored in the System Segment. Its keys list the names of the configuration parameters available to a session. Its values are only used internally in GemStone, to locate the values of the parameters themselves for an individual session.

ErrorSymbols. This SymbolDictionary is defined in the Globals dictionary, and is stored in the System Segment. It maps mnemonic symbols to error numbers.

GemStoneError. This SymbolDictionary is defined in the Globals dictionary, and is stored in the DataCurator Segment. Each key is a Symbol representing a native language, and is associated with an Array of error messages in that language.

Initially, this dictionary contains the single key #English.

InstancesDisallowed. This IdentitySet is defined in the Globals dictionary, and is stored in the System Segment. It is a collection of the GemStone classes for which you can not create instances. Some of these classes (like System) have no instances at all. A fresh GemStone installation already contains all possible instances of others (like Boolean and UndefinedObject).

Table 1.1 Initial Contents of the Globals Dictionary

	Key	The object's class
Numeric Constants	#DecimalMinusInfinity	DecimalFloat
	#DecimalMinusQuietNaN	DecimalFloat
	#DecimalMinusSignalingNaN	DecimalFloat
	#DecimalPlusInfinity	DecimalFloat
	#DecimalPlusQuietNaN	DecimalFloat
	#DecimalPlusSignalingNaN	DecimalFloat
	#MinusInfinity	Float
	#MinusQuietNaN	Float
	#MinusSignalingNaN	Float
	#PlusInfinity	Float
	#PlusQuietNaN	Float
	#PlusSignalingNaN	Float
Non-numeric Constants	#false	Boolean
	#nil	UndefinedObject
	#true	Boolean
Repository and Segments	#DataCuratorSegment	Segment
	#DbfHistory	String
	#NativeLanguage	Symbol
	#PublishedSegment	Segment
	#SystemRepository	Repository
	#SystemSegment	Segment
Collections	#AllClusterBuckets	ClusterBucketArray
	#AllGroups	CanonicalStringDictionary
	#AllSymbols	CanonicalStringDictionary
	#AllUsers	UserProfileSet
	#ConfigurationParameterDict	SymbolKeyValueDictionary
	#ErrorSymbols	SymbolDictionary
	#GemStoneError	SymbolDictionary
	#Globals	SymbolDictionary
	#InstancesDisallowed	IdentitySet

Table 1.1 Initial Contents of the Globals Dictionary

	Key	The object's class
GemStone Internal Objects	#AsciiCollatingTable	ByteArray
	#ConversionDict	SymbolDictionary
	#ConversionReservedOopMap	Array
	#DbConversionStatus	Array
	#DoubleByteAsciiCollatingTable	DoubleByteString
	#GcCandidates	RcQueue
	#GcCandidatesCount	RcCounter
	#GcHints	UndefinedObject
	#GsIndexingSegment	Segment
	#GcWeakReferences	Array
	#ImageVersion	SymbolDictionary
	#OldAllUsers	AbstractUserProfileSet
	#_remoteNil	UndefinedObject
	#SecurityDataSegment	Segment
	#SharedDependencyLists	DepListTable
	#VersionParameterDict	SymbolKeyValueDictionary
plus all kernel classes		

Chapter 2

Class Reference

This chapter describes each of the GemStone kernel classes.

Format of a Class Description

Each description begins on a new page, and has the following form:

Class Name

The class description begins with one or more paragraphs that briefly describe the class's function.

Superclasses This heading lists the class's superclass chain, in order, beginning with the immediate superclass and ending with the class Object. Refer to Figure 1.1 for a graphic representation of the entire kernel class hierarchy.

Class Variables This heading lists the class's class variables (if any). Class variables are shared by a class and all of its instances. For an example, see the descriptions of DateTime and UserProfile in the following pages.

Class Instance Variables This heading lists the class's class instance variables (if any). Class instance variable names are inherited from superclasses, but their values are specific to each class.

Named Instance Variables This heading lists the class's named instance variables (if any). In addition, any named instance variables inherited from superclasses are listed here (and appear first in the list).

Instance Format This heading tells whether instances of the class are pointer or byte objects, non-sequenceable collection objects (bags or sets), or special; whether they are indexable; and whether they are invariant.

Subclass Creation This heading tells whether subclass creation is allowed or disallowed.

Instance Protocol

This heading lists the selectors for messages understood by instances of the class, along with any changes in semantics for inherited methods. Instance methods are arranged by method categories.

Class Protocol

This heading lists the selectors for messages understood by the class object itself, along with any changes in semantics for inherited methods. Class methods are arranged by method categories.

Terminology

The receiver is the object to which a method applies. Think of a method selector and its arguments as a **message** being sent to the receiver. If a method's description does not specify otherwise, the method **returns the receiver**.

Subclass responsibility describes a method that must be defined by subclasses. For example, class Number declares the arithmetic methods `+`, `−`, `*`, and `/` as subclass responsibilities. Each subclass of number, such as Integer, Float, and Fraction, must define these methods in its own way. The class description specifies the method's expected behavior, which the subclass implementation should follow.

A kind of a class is an instance of that class or an instance of one of its subclasses. For example, a function that "returns a kind of Number" returns an object whose class is Number or a subclass of Number.

Disallowed methods are methods defined in a superclass that do not apply to the class being described. Some descriptions suggest alternatives to disallowed methods.

If a method has been optimized to improve performance, its invocation sequence may bypass the GemStone Smalltalk virtual machine. The name of such a method is either a **reserved selector** or an **optimized selector**. You cannot reimplement a reserved selector in any class, and you cannot reimplement an optimized selector in the class within which it is defined. Reserved and optimized selectors are noted in the method descriptions. Refer to the *GemStone Programming Guide* for a complete list of such selectors.

A **class** is **modifiable** if instance variables and class variables can be added or removed.

An **instance** of a class is **variant** if it can be changed (by methods such as `size:` or `add:`, for example).

AbstractCharacter

AbstractCharacter is an abstract superclass that defines behavior common to one-byte and two-byte character classes. Its concrete subclasses include Character and JISCharacter.

Superclasses Magnitude, Object

Named Instance Variables None

Instance Format Byte, Indexable, Invariant

Subclass Creation Allowed

Instance Protocol

Comparing

`<=` *aCharacter* Returns true if the receiver is less than or equal to the argument, otherwise returns false.

`>` *aCharacter* Returns true if the receiver is greater than the argument, otherwise returns false.

`>=` *aCharacter* Returns true if the receiver is greater than the argument, otherwise returns false.

`hash` Returns a numeric hash key for the receiver.

Converting

`asCharacter` (Subclass responsibility.) Returns the Character corresponding to the receiver.

`asInteger` (Subclass responsibility.) Returns the internal code of the receiver.

`asJISCharacter` (Subclass responsibility.) Returns the JISCharacter corresponding to the receiver.

`asLowercase` (Subclass responsibility.) Returns a kind of AbstractCharacter that is the lowercase character corresponding to the receiver. If the receiver is lowercase or has no case, this returns the receiver itself.

`asUppercase` (Subclass responsibility.) Returns a kind of AbstractCharacter that is the uppercase character corresponding to the receiver. If the receiver is uppercase or has no case, this returns the receiver itself.

`digitValue` (Subclass responsibility.) Returns a SmallInteger representing the value of the receiver, a digit, or returns nil if the receiver is not a digit.

Formatting

`displayWidth` (Subclass responsibility.) Returns the width necessary to display the receiver.

Other Comparisons

`asciiLessThan:` *aChar* Returns true if the ASCII code of the receiver is less than that of aCharacter.

Testing

`isDigit` (Subclass responsibility.) Returns true if the receiver is a digit. Returns false otherwise.

`isEquivalent:` *aCharacter*

(Subclass responsibility.) Returns true if the receiver is equivalent to *aCharacter*. The receiver is equivalent to *aCharacter* if the receiver is the same character as the argument regardless of case or internal representation.

`isLowercase` (Subclass responsibility.) Returns true if the receiver is a lowercase character. Returns false otherwise.

`isUppercase` (Subclass responsibility.) Returns true if the receiver is a uppercase character. Returns false otherwise.

Class Protocol

Instance Creation

`fromStream:` *aStream* (Subclass responsibility.) Returns an instance of the receiver's class that is the next character in the stream, *aStream*.

`withValue:` *anInteger* (Subclass responsibility.) Returns an instance of the receiver's class that has the specified internal value.

Non-Printable Characters

`backspace` (Subclass responsibility.) Returns the backspace character from the receiver's character set.

`cr` (Subclass responsibility.) Returns the carriage return character from the receiver's character set.

`esc` (Subclass responsibility.) Returns the escape character from the receiver's character set.

`lf` (Subclass responsibility.) Returns the linefeed character from the receiver's character set.

`newPage` (Subclass responsibility.) Returns the new page character from the receiver's character set.

`space` (Subclass responsibility.) Returns the space character from the receiver's character set.

`tab` (Subclass responsibility.) Returns the tab character from the receiver's character set.

AbstractCollisionBucket

An AbstractCollisionBucket is an Array that is used in a KeyValueDictionary to store a collection of key/value pairs for which the keys hash to the same value.

Superclasses Array, SequenceableCollection, Collection, Object

Named Instance Variables **numElements —** A SmallInteger that gives the number of key/value pairs in the bucket.

Instance Format Pointer, Indexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`at:` *aKey*
Returns the value that corresponds to *aKey*.

`at:` *aKey* `ifAbsent:` *aBlock*
Returns the value that corresponds *aKey*. If no such key/value pair exists, returns the result of evaluating the zero-argument block *aBlock*.

`at:` *aKey* `otherwise:` *aValue*
Returns the value that corresponds to *aKey* by searching the elements in the bucket. If no such key/value pair exists, returns the given alternate value.

`keyAt:` *index*
Returns the key at the specified index.

`keyAt:` *aKey* `otherwise:` *aValue*
Returns the key that corresponds to *aKey* by searching the elements in the bucket. If no such key/value pair exists, returns the given alternate value.

`keyValueDictionary`
Returns nil. Only IdentityCollisionBuckets have the **keyValueDictionary** instance variable.

`numElements`
Returns value of the **numElements** instance variable. (The name **numElements** is provided for compatibility with earlier releases. The instance method `size` is preferred for new code.)

`size`
Returns value of the **numElements** instance variable.

`tableSize` Returns the number of key/value pairs in the capacity of the receiver.

`valueAt:` *index* Returns the value at the specified index.

Adding

`add:` *anAssociation* Adds the key/value pair found in the association to the AbstractCollisionBucket.

Backward Compatibility

Methods in this category are obsolete and are provided only for compatibility with earlier releases of GemStone. They will be removed in a future release.

`doKeys:` *aBlock* Obsolete in GemStone 5.0. Use the `keysDo:` method instead.

`doValues:` *aBlock* Obsolete in GemStone 5.0. Use the `valuesDo:` method instead.

Enumerating

`keysAndValuesDo:` *aBlock*
For each key/value pair in the receiver, evaluates the two-argument block *aBlock* with the key and value as the arguments. Returns the receiver.

`keysDo:` *aBlock* For each key/value pair in the receiver, evaluates the one-argument block *aBlock* with the key as the argument. Returns the receiver.

`valuesDo:` *aBlock* For each key/value pair in the receiver, evaluates the one-argument block *aBlock* with the value as the argument. Returns the receiver.

Formatting

`printOn:` *aStream* Puts a displayable representation of the receiver on the given stream.

Initializing

`initialize` Initializes the instance variable of the receiver to be an empty collisionBucket.

Removing

`removeKey:` *aKey* `ifAbsent:` *aBlock*
: Removes the key/value pair having the key *aKey*. If *aKey* is not found, returns the result of evaluting the zero-argument block *aBlock*.

Searching

`firstPair`
: Returns an association containing the receiver's first key/value pair. If the receiver is empty, returns an association containing nils.

`includesKey:` *aKey*
: Returns true if the receiver contains a key that is equal to *aKey*. Otherwise, returns false.

`searchForKey:` *aKey*
: Returns the index of *aKey*, or if not found, nil.

Updating

`at:` *aKey* `put:` *aValue*
: Stores the aKey/aValue pair in the receiver. Returns *aValue*.

`at:` *aKey* `put:` *aValue* `keyValDict:` *aKeyValDict*
: Stores the aKey/aValue pair in the receiver. Returns *aValue*.

`at:` *anIndex* `putKey:` *aKey*
: Stores the key *aKey* into the key part of the key/value pair referenced by *anIndex*. Note that this method overwrites the key value at the given index and destroys the ordering of the keys in the receiver. Returns *aKey*.

`at:` *anIndex* `putValue:` *aValue*
: Stores the value *aValue* into the value part of the key/value pair referenced by atIndex. Returns *aValue*.

`keyValueDictionary:` *aDict*
: No-op for AbstractCollisionBuckets. Used only for IdentityCollisionBuckets.

Class Protocol

Instance Creation

`new`	Returns an AbstractCollisionBucket with a default capacity of four key/value pairs.
`new:` *aSize*	Returns an AbstractCollisionBucket with the specified size.

AbstractDictionary

AbstractDictionary is an abstract class that provides the protocol for collections whose elements can be accessed by an associated lookup key. Concrete classes of AbstractDictionary store the key-value pairs either directly or as Associations. See the documentation for the Dictionary and KeyValueDictionary classes for more information.

Superclasses	Collection, Object
Named Instance Variables	None
Instance Format	Pointer, Indexable, Variant
Subclass Creation	Allowed

Instance Protocol

Accessing

`at:` *aKey*
Returns the value of the Association with key *aKey*. Generates an error if no such key exists.

`at:` *aKey* `ifAbsent:` *aBlock*
Returns the value associated with key *aKey*. If no such key/value pair or association exists, returns the result of evaluating the zero-argument block *aBlock*.

`at:` *aKey* `otherwise:` *value*
Returns the value that corresponds to *aKey*. If no such key/value pair or association exists, returns the given alternate *value*.

`keyAtValue:` *anObject*
Returns the key of the first value equal to *anObject*. If no match is found, runtime error objErrNotInColl is signalled.

`keyAtValue:` *anObject* `ifAbsent:` *aBlock*
Returns the key of the first value equal to the given object, *anObject*. If no match is found, evaluates and returns the result of the block *aBlock*.

`keys`
Returns a Set containing the receiver's keys.

`size`
Returns the number of elements (Associations/key-value pairs) contained in the receiver.

`values` Returns an OrderedCollection containing the receiver's values.

Adding

`add:` *anAssociation* Adds the association or the key-value pair contained in the association to the receiver. If the receiver already includes an association/key-value pair whose key is equal to that of *anAssociation*, then this method redefines the value portion of that Association/key-value pair. Returns *anAssociation*.

`addAll:` *aCollection* Adds to the receiver all the associations or key-value pairs contained in *aCollection*. *aCollection* must be a collection of Associations or a dictionary. Returns the argument, *aCollection*.

Backward Compatibility

Methods in this category are obsolete and are provided only for compatibility with earlier releases of GemStone. They will be removed in a future release.

`collectValues:` *aBlock* Obsolete in GemStone 5.0. Use the `collect:` method instead.

`detectAssociations:` *aBlock*
Obsolete in GemStone 5.0. Use the `associationsDetect:` method instead.

`detectAssociations:` *aBlock* `ifNone:` *exceptionBlock*
Obsolete in GemStone 5.0. Use the `associationsDetect:ifNone:` method instead.

`detectValues:` *aBlock* Obsolete in GemStone 5.0. Use the `keysAndValuesDo:` method instead.

`detectValues:` *aBlock* `ifNone:` *exceptionBlock*
Obsolete in GemStone 5.0. Use the `keysAndValuesDo:` method instead.

`doAssociations:` *aBlock*
Obsolete in GemStone 5.0. Use the `associationsDo:` method instead.

`doKeys:` *aBlock* Obsolete in GemStone 5.0. Use the `keysDo:` method instead.

`doKeysAndValues:` *aBlock*
Obsolete in GemStone 5.0. Use the `keysAndValuesDo:` method instead.

`doValues:` *aBlock* Obsolete in GemStone 5.0. Use the `valuesDo:` method instead.

`includesValue:` *aValue* Obsolete in GemStone 5.0. Use the `includes:` method instead.

`occurrencesOfValue:` *aValue*
Obsolete in GemStone 5.0. Use the `occurrencesOf:` method instead.

`rejectValues:` *aBlock* Obsolete in GemStone 5.0. Use the `reject:` method instead.

`removeKeys:` *keys* Obsolete in GemStone 5.0. Use the `removeAllKeys:` method instead.

`selectValues:` *aBlock* Obsolete in GemStone 5.0. Use the `select:` method instead.

`selectValuesAsArray:` *aBlock*
Obsolete in GemStone 5.0.

Comparing

= *anAbstractDictionary* Returns true if all of the following conditions are true:

1. The receiver and *anAbstractDictionary* are of the same class.
2. The two dictionaries are of the same size.
3. The corresponding keys and values of the receiver and *anAbstractDictionary* are equal.

`hash` Returns a numeric hash key for the receiver.

Enumerating

`associationsDetect:` *aBlock*

Evaluates *aBlock* repeatedly, with the associations of the receiver as the argument. Returns the first association for which the block evaluates to true when the association is used as the argument to the block. If none of the receiver's associations evaluates to true, generates an error. The argument *aBlock* must be a one-argument block.

`associationsDetect:` *aBlock* `ifNone:` *exceptionBlock*

Evaluates *aBlock* repeatedly, with the associations of the receiver as the argument. Returns the first association for which *aBlock* evaluates to true when the association is used as the argument to the block. If none of the receiver's associations evaluates to true, this method evaluates the argument *exceptionBlock* and returns its value. The argument *aBlock* must be a one-argument block, and *exceptionBlock* must be a zero-argument block.

`associationsDo:` *aBlock*

Iteratively evaluates the one argument block, *aBlock*, using each association in the receiver as the argument to the block. If the receiver stores the key-value pairs directly, new associations are created for the key-value pairs and used as the arguments to *aBlock*. Returns the receiver.

`collect:` *aBlock* Evaluates *aBlock* with each of the receiver's values as the argument and collects the resulting values into the appropriate Dictionary at the corresponding key values.

`collectAssociations:` *aBlock*

Evaluates *aBlock* with each of the receiver's Associations (or Associations created using the key-value pairs) as the argument and collects the resulting values into an Array. Returns the newly created Array.

`collectValuesAsArray:` *aBlock*

Evaluates *aBlock* with each of the receiver's values as the argument and collects the resulting values into an Array. Returns the new Array.

`do:` *aBlock*
Iteratively evaluates the one argument block, *aBlock*, using the value part of each Association or key-value pair as the argument of the block. Returns the receiver.

`keysAndValuesDo:` *aBlock*
Iteratively evaluates the two argument block, *aBlock*, using each key and value of the receiver as the argument to the block. Returns the receiver.

`keysDo:` *aBlock*
Iteratively evaluates the one argument block, *aBlock*, using each key of the receiver as the argument to the block. Returns the receiver.

`reject:` *aBlock*
Evaluates *aBlock* with each of the receiver's values as the argument. Stores the key-value pairs for which *aBlock* is false into a dictionary of the same class as the receiver, and returns the new dictionary. The argument *aBlock* must be a one-argument block.

`rejectAssociations:` *aBlock*
Evaluates *aBlock* with each of the receiver's elements as the argument. Stores the values for which *aBlock* is false into a collection of the same class as the receiver, and returns the new collection. The argument *aBlock* must be a one-argument block. Uses associative access when the argument is a SelectionBlock.

`rejectValuesAsArray:` *aBlock*
Evaluates *aBlock* with each of the receiver's values as the argument and returns an array containing the values for which *aBlock* evaluates false.

`select:` *aBlock*
Evaluates *aBlock* with each of the receiver's values as the argument. Stores the values for which *aBlock* is true into the dictionary of the same class as the receiver, and returns the new dictionary. The argument *aBlock* must be a one-argument block.

`selectAssociations:` *aBlock*
Evaluates *aBlock* with each of the receiver's elements as the argument. Stores the values for which *aBlock* is true into a collection of the same class as the receiver, and returns the new collection. The argument *aBlock* must be a one-argument block. Uses associative access when the argument is a SelectionBlock.

`speciesForCollect`	Returns a class, an instance of which should be used as the result of `collect:` or other projections applied to the receiver.
`speciesForSelect`	Returns a class, an instance of which should be used as the result of `select:` or other projections applied to the receiver.
`valuesDo:` *aBlock*	Iteratively evaluates the one argument block, *aBlock*, using each value of the receiver as the argument to the block. Returns the receiver.

Formatting

`printOn:` *aStream*	Puts a displayable representation of the receiver on the given stream.

Hashing

`hashFunction:` *aKey*	The `hash` function should perform some operation on the value of the key (*aKey*) which returns a value in the range `1..tableSize`.

Removing

`remove:` *anObject*	Disallowed. Use `removeKey:` instead.
`remove:` *anObject* `ifAbsent:` *anExceptionBlock*	Disallowed. Use `removeKey:ifAbsent:` instead.
`removeAll:` *aCollection*	Disallowed. Use `removeAllKeys:` instead.
`removeAllIdentical:` *aCollection*	Disallowed.
`removeAllKeys:` *keys*	Removes all the keys equal to the given keys from the receiver. An error is not generated if keys equal to any of the specified keys are not present. Returns the collection *keys*.
`removeAllKeys:` *keys* `ifAbsent:` *aBlock*	Removes all the keys equal to the given keys from the receiver and returns the collection *keys*. For any key which is not a valid key of the receiver, *aBlock* is evaluated with the key as the argument.
`removeIdentical:` *anObject*	Disallowed.

`removeIdentical:` *anObject* `ifAbsent:` *anExceptionBlock*
Disallowed.

`removeKey:` *aKey* Removes the Association or key-value pair with key equal to *aKey* from the receiver and returns the value portion of that Association or key-value pair respectively. If no Association is present with key equal to *aKey*, reports an error.

`removeKey:` *aKey* `ifAbsent:` *aBlock*
Removes the Association or key-value pair with key equal to *aKey* from the receiver and returns the value associated with the Association or the key-value pair. If no Association or key-value pair is present with key equal to *aKey*, evaluates the zero-argument block *aBlock* and returns the result of that evaluation.

Searching

`includes:` *aValue* Returns true if the receiver contains a value that is equal to *aValue*. Returns false otherwise.

`includesAssociation:` *anAssociation*
Returns true if *anAssociation* is equal to one of the Associations of the receiver. Returns false otherwise.

`includesIdentical:` *aValue*
Returns true if the receiver contains a value that is identical to *aValue*. Returns false otherwise.

`includesIdenticalAssociation:` *anAssociation*
Returns true if *anAssociation* is identical to one of the Associations of the receiver. Returns false otherwise.

`includesKey:` *aKey* Returns true if the receiver contains an Association or a key-value pair whose key is equal to *aKey*. Returns false otherwise.

`occurrencesOf:` *aValue* Returns the number of Associations or key-value pairs in the receiver with value equal to *aValue*.

`occurrencesOfIdentical:` *aValue*
Returns the number of Associations or key-value pairs in the receiver with a value that is identical to *aValue*.

Sorting

`sortAscending:` *aSortSpec*

Returns an Array containing Associations constructed from the receiver, sorted in ascending order, as determined by the values of the instance variables represented by *aSortSpec*.

See `UnorderedCollection | sortAscending:` for further documentation.

`sortDescending:` *aSortSpec*

Returns an Array containing Associations constructed from the receiver, sorted in descending order, as determined by the values of the instance variables represented by *aSortSpec*.

See `UnorderedCollection | sortDescending:` for further documentation.

`sortWith:` *aSortPairArray*

Returns an Array containing Associations constructed from the receiver, sorted according to the contents of *aSortPairArray*.

See `Collection | sortWith:` for further documentation.

Storing and Loading

`loadFrom:` *passiveObj* `size:` *varyingSize*

Reads from *passiveObj* the passive form of an object. Converts the object to its active form by loading the information into the receiver.

Updating

`at:` *aKey* `put:` *aValue*

Creates a new Association with the given key and value and adds it to the receiver or adds the key-value pair to the receiver depending on the class of the receiver. If the receiver already contains an Association or key-value pair with the given key, this method makes *aValue* the value of that Association or key-value pair. Returns *aValue*.

`size:` *newSize*

Disallowed. You should not change the size of a dictionary explicitly.

Class Protocol

Instance Creation

`new`	(Subclass responsibility.) Returns an instance of the receiver.
`new:` *anInteger*	(Subclass responsibility.) Returns an instance of the receiver.

Storing and Loading

`loadFrom:` *passiveObj*	Reads from *passiveObj* the passive form of an object. Converts the object to its active form by loading the information into a new instance of the receiver. Returns the new instance.

AbstractSession

AbstractSession is an abstract class for describing within GemStone sessions that exist either in GemStone or in some other server software. It is intended to provide support in GemStone for two-phase commit protocols between transactions in related sessions.

For example, a session external to GemStone could be a session in an external database. It could be spawned by the current GemStone session. The GemStone session object permits access to its symbol list for name resolution within its name space, enables execution of Smalltalk code within the session, and allows control of its transaction state.

Superclasses	Object
Named Instance Variables	None
Instance Format	Pointer, Nonindexable, Variant
Subclass Creation	Allowed

AbstractUserProfileSet

An AbstractUserProfileSet is an IdentitySet whose elements must be instances of class UserProfile. You may not create subclasses or instances of AbstractUserProfileSet. Only one instance of AbstractUserProfileSet is permitted in a GemStone repository that was upgraded from GemStone 4.1.

Superclasses IdentitySet, IdentityBag, UnorderedCollection, Collection, Object

Named Instance Variables None

Instance Format Nsc, Nonindexable, Variant

Subclass Creation Disallowed

Instance Protocol

Accessing

`userWithId:` *aString* Searches the receiver for a UserProfile whose userId is equal to *aString*, and returns that UserProfile. Generates an error if no userId is equal to *aString*.

`userWithId:` *aString* `ifAbsent:` *aBlock*

Searches the receiver for a UserProfile whose userId is equal to *aString*, and returns that UserProfile. Evaluates the argument *aBlock* if no userId is equal to *aString*.

Adding

`add:` *aUserProfile* (Subclass responsibility.) Adds *aUserProfile* to the receiver.

`addAll:` *aCollection* Reimplemented to maintain KeyValueDictionary on AllUsers.

Removing

`remove:` *anObject* Reimplemented to maintain KeyValueDictionary on AllUsers.

`remove:` *anObject* `ifAbsent:` *aBlock*
Reimplemented to maintain KeyValueDictionary on AllUsers.

`removeAll:` *aCollection* Reimplemented to maintain KeyValueDictionary on AllUsers.

`removeAllPresent:` *aCollection*
Reimplemented to maintain KeyValueDictionary on AllUsers.

`removeIfPresent:` *anObject*
Reimplemented to maintain KeyValueDictionary on AllUsers.

Array

Array is a concrete subclass of SequenceableCollection that capitalizes upon the indexability of its elements. An Array permits its elements to be placed in any order, but once placed, it retains the order until changed explicitly. Integer indexes are often used to access elements directly, randomly, or in alternate or unpredictable orders not necessarily related to the sequence of the elements as placed in the collection. Thus, an index is often used as the address for an element.

Uninitialized Array elements are nil.

Superclasses SequenceableCollection, Collection, Object

Named Instance Variables None

Instance Format Pointer, Indexable, Variant

Subclass Creation Allowed

Instance Protocol

Adding

`addLast:` *newObject* Adds *newObject* as the last element of the receiver and returns *newObject*.

Comparing

`hasIdenticalContents:` *anArray*

Returns true if all of the following conditions are true:

1. The receiver and *anArray* are of the same class.
2. The two arrays are the same size.
3. The corresponding elements of the receiver and *anArray* are equal.

Returns false otherwise.

Enumerating

`speciesForCollect` Returns a class, an instance of which should be used as the result of `collect:` or other projections applied to the receiver.

Class Protocol

Subclass Creation

`byteSubclass:` *aString* `classVars:` *anArrayOfClassVars*
`classInstVars:` *anArrayOfClassInstVars*
`poolDictionaries:` *anArrayOfPoolDicts* `inDictionary:` *aDictionary*
`instancesInvariant:` *invarBoolean*

Disallowed for Array and its subclasses.

`byteSubclass:` *aString* `classVars:` *anArrayOfClassVars*
`classInstVars:` *anArrayOfClassInstVars*
`poolDictionaries:` *anArrayOfPoolDicts* `inDictionary:` *aDictionary*
`newVersionOf:` *oldClass* `instancesInvariant:` *invarBoolean*

Disallowed for Array and its subclasses.

Association

An Association is a pair of associated objects: a key and a value. A Dictionary is a collection of Associations; thus, much of the protocol that affects Associations is actually defined for instances of Dictionary. See the description of Dictionary for details.

Superclasses Object

Named Instance Variables **key —** An object, usually used as a reference to its **value**.

value — Another object, referenced by its **key**.

Instance Format Pointer, Nonindexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`key` Returns the **value** of the receiver's **key**.

`value` Returns the **value** portion of the receiver.

Clustering

`clusterDepthFirst` This method clusters the receiver, its **key**, and its **value** in depth-first order. It returns true if the receiver has already been clustered during the current transaction, false otherwise.

Comparing

< *anObject* Returns true if the **key** of the receiver collates before *anObject*. Returns false otherwise.

<= *anObject* Returns true if the **key** of the receiver collates before *anObject*, or if the **key** of the receiver is equivalent to *anObject*. Returns false otherwise.

= *anObject* Returns true if (a) the receiver and *anObject* are of the same class, (b) the receiver and *anObject* have equal keys and (c) the receiver and *anObject* have the same **value**. Returns false otherwise.

> *anObject* Returns true if the **key** of the receiver collates after *anObject*. Returns false otherwise.

`>=` *anObject* — Returns true if the **key** of the receiver collates after *anObject*, or if the **key** of the receiver is equivalent to *anObject*. Returns false otherwise.

`hash` — Returns an Integer hash code for the receiver.

`~=` *anObject* — Returns true if one or more of the conditions specified in = method are not satisfied. Returns false otherwise.

Formatting

`printOn:` *aStream* — Puts a displayable representation of the receiver on the given stream.

Updating

`key:` *aKey* — Sets the object *aKey* as the **key** of the receiver.

`key:` *aKey* `value:` *aValue*
Sets the object *aKey* as the **key** of the receiver, and the object *aValue* as the **value** of the receiver.

`value:` *aValue* — Sets the object *aValue* as the **value** of the receiver.

Class Protocol

Instance Creation

`newWithKey:` *aKey* `value:` *aValue*
Returns a new Association with the argument *aKey* as its key and with *aValue* as its value.

AutoComplete

The AutoComplete class is a name completer. Given a collection of names it can find a prefix of one fairly quickly.

Superclasses Object

Named Instance Variables

realStrings — A SortedCollection of the StringPairs holding the original strings and a mapping to **lookupStrings**.

lookupStrings — The uppercase versions of all the strings in the search domain.

Instance Format Pointer, Nonindexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`strings` Returns the list of real strings.

Completing

`commonChars:` *s1* `with:` *s2*

Returns a count of the number of characters common between two strings.

`complete:` *string* Attempts to complete the given string from our current set. Returns either the original string, or a replacement for it that is either the same length or longer.

Initializing

`addString:` *str*	Adds the given string to the search domain of the receiver.
`stringPairSet:` *alist*	Sets up information needed to do string completion on the given set of strings. The strings are already in our desired StringPairSet form.
`strings:` *strings*	Sets up information needed to do string completion on the given set of strings.
`strings:` *strings* `cluster:` *cluster*	Sets up information needed to do string completion on the given set of strings.

Bag

A Bag is an UnorderedCollection in which any distinct object can occur any number of times. Adding the same (identical) object to a Bag multiple times simply causes it to occur multiple times in the Bag.

Since a Bag is an equality-based collection, different (non-identical) but equivalent (equal) objects are not treated as distinct from each other. In IdentityBags, they are distinct. Adding multiple equivalent objects to a Bag yields a Bag with multiple occurrences of the object that was added last.

You can create subclasses of Bag to restrict the kind of elements it contains. When creating a subclass of Bag, you must specify a class as the aConstraint argument. This class is called the element kind of the new subclass. For each instance of the new subclass, the class of each element must be of the element kind.

Superclasses UnorderedCollection, Collection, Object

Named Instance Variables **dict —** A KeyValueDictionary that organizes the elements and element counts for the Bag.

size — For GemStone internal use.

Instance Format Nsc, Nonindexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`at:` *anIndex* Disallowed.

`size` Returns the number of elements contained in the receiver.

Adding

`add:` *newObject* Makes *newObject* one of the receiver's elements and returns *newObject*.

`add:` *anObject* `withOccurrences:` *anInteger*
Adds *anObject anInteger* number of times to the receiver and returns *anObject*.

Removing

`removeAll:` *aCollection* Removes one occurrence of each element of *aCollection* from the receiver and returns the receiver. Generates an error if any element of *aCollection* is not present in the receiver.

Searching

`includesIdentical:` *anObject*

Returns true if *anObject* is identical to one of the elements of the receiver. Returns false otherwise.

Updating

`at:` *anIndex* `put:` *anObject*

Disallowed.

`changeToSegment:` *segment*

Assigns the receiver and its private objects to the given segment.

Class Protocol

Instance Creation

`new` Returns an instance of the receiver whose contents are empty.

`new:` *initialSize* Returns an instance of the receiver whose contents are empty.

Behavior

Behavior is an abstract superclass with two concrete subclasses: Metaclass and Class. You may not create any other subclasses of Behavior.

Behavior describes the protocol common to all instances of Class and Metaclass. In other words, you can send the messages listed here to any Class or Metaclass. In the method descriptions below, "superclass" refers to the superclass of instances of the receiver, not to the superclass of the receiver itself.

Superclasses

Object

Named Instance Variables

superClass — The Behavior's immediate superclass in the class hierarchy (a Metaclass).

format — A SmallInteger that encodes the following information in its bits:

- (format \\ 4) = 0 if instances of the behavior are pointer objects.
- (format \\ 4) = 1 if instances of the behavior are byte objects.
- (format \\ 4) = 2 if instances of the behavior are non-sequenceable collections (NSCs, such as Bags and Sets).
- (format \\ 4) = 3 if instances of the behavior are special objects.
- ((format // 4) \\ 2) = 1 if instances of the behavior are indexable.
- ((format // 8) \\ 2) = 1 if instances of the behavior are invariant.
- ((format // 16) \\ 2) = 1 if the behavior has **constraints** upon what can be stored in its instance variables.
- ((format // 32) \\ 2) = 1 the behavior does not allow subclass creation.
- ((format // 128) \\ 2) = 1 the behavior does not allow structural access from GemBuilder for C.

instVars — A SmallInteger telling the number of instance variables in instances of this class (including those inherited from superclasses). Each instance of Behavior is limited to 255 named instance variables.

instVarNames — An invariant Array of Symbols giving the names of the Behavior's instance variables, including those inherited from superclasses. Each instance variable name is limited to 64 characters, and must begin with an alphabetic character or an underscore ("_"). For more information, see the *GemStone Programming Guide*.

constraints — An invariant Array of Classes. Each element in the Array is the classkind of a corresponding instance variable defined in a class or inherited from a superclass.

classVars — A SymbolDictionary used when compiling methods in this Behavior. Each instance of a class has its own instance variables, which may differ in value. Each class has its own class variables, which have the same value for all instances of the class. For each SymbolAssociation in this dictionary, the key is a Symbol representing a class variable, and the corresponding value is the value of that class variable. Each class variable name is limited to 64 characters, and must begin with an alphabetic character or an underscore ("_").

methodDict — A GsMethodDictionary that has all of the additional protocol (not inherited from superclasses) for instances of this Behavior.

poolDictionaries — An Array of SymbolDictionaries used when compiling methods in this Behavior. The dictionaries contain objects that can be shared by multiple classes and multiple users.

categories — A GsMethodDictionary that categorizes selectors in this Behavior. For each element in this dictionary, the key is a method category Symbol, and the corresponding value is a SymbolSet of the selectors for that method category.

secondarySuperclasses — Reserved for future use by GemStone Systems, Inc.

Instance Format Pointer, Nonindexable, Variant

Subclass Creation Disallowed

Instance Protocol

Accessing Categories

`categoryNames` Returns an Array of Symbols. The elements of the Array are the receiver's category names (excluding names inherited from superclasses).

`selectorsIn:` *categoryName*
Returns an Array of all selectors in the specified category. If *categoryName* is not in the receiver's method dictionary, generates an error.

`sortedCategoryNames` Returns an Array of Symbols. The elements of the collection are the receiver's category names (excluding names inherited from superclasses).

`sortedSelectorsIn:` *categoryName*
Returns an Array of all selectors in the specified category, sorted in ascending order.

Accessing Constraints

`allConstraints` If the receiver defines a non-sequenceable collection (bag or set) class, this returns a single Class, the element kind of the receiver.

Otherwise, this returns an Array of Classes. Each element in that Array is the classkind of a corresponding instance variable. The ordering of the elements in the Array is the same as the ordering of instance variables in the receiver.

`constraintOn:` *aString* Returns the classkind constraint for the instance variable represented by *aString*. If the instance variable named *aString* is not constrained, returns Object. If no instance variable named *aString* exists for objects whose Behavior is defined by the receiver, returns nil.

`elementConstraint` Returns the kind of objects that instances of the receiver can hold.

`varyingConstraint`	Returns the constraint on the unnamed part of the receiver (a kind of Class). If the receiver has no constraint on its unnamed part, or if it has no unnamed part, this method returns Object.

Accessing the Class Format

`firstPublicInstVar`	Returns the index of the first publicly available instance variable storage location, whether or not a public instance variable has actually been defined.
`format`	Returns the value of the **format** instance variable.
`hasPublicInstVars`	Returns true if the receiver has publicly-visible instance variables.
`implementationFormat`	Returns the three least-significant bits of the receiver's **format** instance variable. The values of those bits mean the following: 0 Oop non-indexable 1 Byte non-indexable 2 NSC non-indexable 3 Special non-indexable 4 Oop indexable 5 Byte indexable
`instancesInvariant`	Returns true if instances of the receiver may not change value after they have been committed to GemStone. Otherwise, returns false.
`instSize`	Returns the number of named instance variables in the receiver, including all inherited instance variables.
`isBytes`	Returns true if instances of the receiver are byte objects. Otherwise, returns false.
`isBytesOrSpecial`	Returns whether instances of the receiver are byte objects.
`isIndexable`	Returns true if instances of the receiver have indexed variables. Otherwise, returns false.
`isNonByteVarying`	Returns true if the instances of the receiver are not byte objects and have unnamed instance variables; returns false otherwise.

`isNsc`	Returns true if instances of the receiver are non-sequenceable collections (UnorderedCollections). Otherwise, returns false.
`isPointers`	Returns true if instances of the receiver are pointer objects. Otherwise, returns false.
`isProtected`	Returns true if instances of the receiver may not be accessed structurally through GemBuilder for C.
`isVariable`	Returns true if instances of the receiver have an unnamed part.
`subclassesDisallowed`	Returns true if subclasses of the receiver have been disallowed by means of `Behavior | disallowSubclasses`. Otherwise, returns false.

Accessing the Class Hierarchy

`allSuperClasses`	Returns an Array of the superclasses of the receiver, beginning with the immediate superclass, and excluding the receiver.
`inheritsFrom:` *aClass*	Returns true if the argument *aClass* is on the receiver's superclass chain; returns false if it isn't.
`superClass`	Returns the receiver's superclass.
`superclass`	Returns the receiver's superclass.

Accessing the Method Dictionary

`allSelectors`	Returns an Array of Symbols, consisting of all of the message selectors that instances of the receiver can understand, including those inherited from superclasses. For keyword messages, the Symbol includes each of the keywords, concatenated together.
`canUnderstand:` *aSelector*	Returns true if the receiver can respond to the message indicated by *aSelector*, returns false otherwise. The selector (a String) can be in the method dictionary of the receiver or any of the receiver's superclasses.

`categoryOfSelector:` *aSelector*

Returns a Symbol which is the name of the category for the specified selector, or nil if the selector was not found in any category.

`compiledMethodAt:` *aSelector*

Returns the compiled method associated with the argument *aSelector* (a String). The argument must be a selector in the receiver's method dictionary; if it is not, this method generates an error.

`includesSelector:` *aString*

Returns true if the receiver defines a method for responding to *aString*.

`selectors` Returns an Array of Symbols, consisting of all of the message selectors defined by the receiver. (Selectors inherited from superclasses are not included.) For keyword messages, the Symbol includes each of the keywords, concatenated together.

`sourceCodeAt:` *aSelector* Returns a String representing the source code for the argument, *aSelector*. If *aSelector* (a String) is not a selector in the receiver's method dictionary, this generates an error.

`whichClassIncludesSelector:` *aString*

If the selector *aString* is in the receiver's method dictionary, returns the receiver. Otherwise, returns the most immediate superclass of the receiver where *aString* is found as a message selector. Returns nil if the selector is not in the method dictionary of the receiver or any of its superclasses.

Accessing Variables

`allClassVarNames` Returns an Array of Symbols, consisting of the names of the class variables addressable by this class, including those inherited from superclasses. Contrast with `classVarNames`.

`allInstVarNames` Returns an Array of Symbols, consisting of the names of all the receiver's instance variables, including those inherited from superclasses. The ordering of the names in the Array follows the ordering of the superclass hierarchy; that is, instance variable names inherited from Object are listed first, and those peculiar to the receiver are last.

`allSharedPools` Returns an Array of pool dictionaries used by this class and its superclasses. Contrast with `sharedPools`.

`classVarAt:` *aClassVar* Returns the value of the class variable *aClassVar*.

`classVarNames` Returns an Array of Symbols naming the class variables defined by this class. Inherited class variables are not included; contrast with `allClassVarNames`.

`constraintOfInstVar:` *aString*
Returns the constraint on the instance variable named *aString* for instances of the receiver. Generates an error if *aString* is not the name of an instance variable defined by the receiver.

`instVarNames` Returns an Array of Symbols naming the instance variables defined by the receiver, but not including those inherited from superclasses. Contrast with `allInstVarNames`.

`offsetOfInstVar:` *aSymbol*
Returns the integer offset at which the instance variable named *aSymbol* is stored in instances of the receiver. Returns zero if the instance variable is not found.

`scopeHas:` *aVariableName* `ifTrue:` *aBlock*
If *aVariableName* (a String) is specified as a variable in the receiver or one of its superclasses, this evaluates the zero-argument block *aBlock* and returns the result of evaluating *aBlock*. Otherwise, returns false.

`sharedPools` Returns an Array of pool dictionaries used by this class. Superclasses are not included; contrast with `allSharedPools`.

Analysis

`referencedStrings` Returns a Set containing all Strings and InvariantStrings referenced by the methods in this class and its metaclass.

Browsing

`copyMethodsFrom:` *sourceClass* `dictionaries:` *dicts*

Copies all instance and class methods from the *sourceClass*. Returns an Array of methods in the source class which failed to compile in this class. Some of them might be class methods. The Array is empty if no methods failed to compile.

`removeSelector:` *aString* `ifAbsent:` *aBlock*

Removes the method whose selector is *aString* from the receiver's method dictionary. If the selector is not in the method dictionary, returns the result of evaluating the zero-argument block *aBlock*. Otherwise, returns the receiver.

Category

`category` Returns the classCategory instance variable of the receiver. If the receiver's category is nil, returns its superclass's category.

Clustering

`clusterBehavior` This method 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 has already been clustered during the current transaction; returns false otherwise.

It is recommended that when several classes are being clustered in a transaction, send `clusterBehavior` to all classes to be clustered, then send `clusterDescription`.

`clusterBehaviorExceptMethods:` *aCollectionOfMethodNames*

This method allows you to cluster the receiver more efficiently by omitting infrequently-used methods from the clustering. The methods that you designate as *aCollectionOfMethodNames* will not be clustered with the receiver. Thus, the methods that are frequently used will be packed more densely. Returns true if the receiver has already been clustered during the current transaction; returns false otherwise.

This method works by first clustering all methods named into the max cluster bucket, preventing them from showing up in the default cluster bucket. It then uses the standard method to cluster behavior.

`clusterDescription` This method clusters, in depth-first order, those instance variables in the receiver that describe the structure of the receiver's instances. The following instance variables are clustered: **instVarNames**, **classVars**, and **categories**. (The receiver itself is not clustered.) Returns true if the receiver has already been clustered during the current transaction; returns false otherwise.

It is recommended that when several classes are being clustered in a transaction, send `clusterBehavior` to all classes to be clustered, then send `clusterDescription`.

Copying

`copy` Disallowed. To create a new Class or Metaclass, use `Class | subclass:instVarNames:..` instead.

Enumerating

`allSuperClassesDo:` *aBlock*

Evaluates *aBlock* with each of the receiver's superclasses as the argument, beginning with the immediate superclass.

Fileout

`fileOutCategories` Returns a string with all the receiver's methods in Topaz Filein format.

`fileOutCategoriesOn:` *stream*
Writes the receiver's categories and methods onto the given stream in Topaz filein format.

`fileOutCategory:` *catName*
Returns a string containing the methods of the given category in Topaz Filein format.

`fileOutCategory:` *catName* `on:` *stream*
Files out the given category on the given stream.

`fileOutClass` Returns a string with the receiver's class definition and all the receiver's methods in Topaz Filein format.

`fileOutClassByCategoryOn:` *stream*
Writes the receiver's definition and methods onto the given stream in filein format.

`fileOutClassOn:` *stream*
Writes the receiver's definition and methods onto the given stream in filein format.

`fileOutHelpOn:` *aStream*
Stubbed in this version. Geode adds help attributes to classes.

`fileOutIconOn:` *stream* Stubbed in this version. Geode adds icon attributes to classes.

`fileOutMethod:` *selector* Returns a string with the given method's category and source in Topaz Filein format.

`fileOutMethod:` *selector* `on:` *stream*
Writes the given method's source to the given *stream* in Topaz Filein format.

`fileOutMethodRemovalOn:` *stream* `name:` *nm*
Writes code to remove all the receiver's methods onto the given *stream* in filein format.

`fileOutMethods` Returns a string with all the receiver's methods in Topaz Filein format.

`fileOutMethodsOn:` *stream*

File out this class's methods, but sort the selectors alphabetically.

`fileOutPostMethodsOn:` *stream*

This method gives classes an opportunity to file out information not normally emitted by the predefined fileout methods. Classes may override this method (as a class method, of course) to add extra code to the output *stream*. Emitted code should be in Topaz filein format. It will be placed after method creation.

`fileOutPreClassOn:` *stream*

This method gives classes an opportunity to file out information not normally emitted by the predefined fileout methods. Classes may override this method (as a class method, of course) to add extra code to the output *stream*. Emitted code should be in Topaz filein format. It will be placed before any other fileout information for the class.

`fileOutPreMethodsOn:` *stream*

This method gives classes an opportunity to file out information not normally emitted by the predefined fileout methods. Classes may override this method (as a class method, of course) to add extra code to the output *stream*. Emitted code should be in Topaz filein format. It will be placed after existing method removal and before method creation.

`nameForFileout` Returns the name to be used for this class for fileout.

Formatting

`asString` Returns a String that indicates the class of the receiver.

Indexing Support

`btreeLeafNodeClass` Returns the class of BtreeLeafNode to create for an equality index whose last object along the path is an instance of the receiver.

`sortNodeClass` Returns the class of SortNode to create for sorting on instances of the receiver.

Instance Creation

`basicNew`	Creates a new, uninitialized instance of the receiver.
`basicNew:` *anInteger*	Creates a new, uninitialized instance of the receiver with the given number of indexed instance variables.
`new`	Returns an instance of the receiver with no indexed variables.
`new:` *anInteger*	Returns an instance of the receiver with the specified number of indexed variables. Generates an error if the receiver is not indexable or if *anInteger* is not a positive SmallInteger. For new byte objects, all indexed variables are set to zero; for new pointer objects, all indexed variables are set to nil.

Modifying Classes

`addInstVar:` *aString*	Adds a new instance variable named aSymbol to the receiver. The argument *aString* must be a valid GemStone Smalltalk identifier and must be distinct from the names of all other instance variables previously defined for the receiver, its superclasses, and its subclasses. The new instance variable becomes the last named instance variable in the receiver, and is inserted at the appropriate position in each of the receiver's subclasses, to preserve the rules for inheritance of instance variables. If, for any of the receiver's subclasses, the new instance variable is not the last named instance variable, then all instance methods for that subclass are recompiled using the symbol list of the current user. If an error occurs during recompilation of methods, the new instance variable will have been added to the receiver and all of its subclasses, but some methods in some subclasses will not have been recompiled. To successfully invoke this method, the receiver must meet one of these two conditions: • The receiver and all of its subclasses must be modifiable. • The receiver must disallow subclasses and must have no unnamed instance variables.

`addInstVar:` *aSymbol* `withConstraint:` *aClass*

Adds a new instance variable named *aSymbol* to the receiver and constrains the value of that variable to be of the kind *aClass*. The instance variable is added in the same way and under the same conditions as described for the `addInstVar:` method.

The argument *aClass* must be a kind of Class; otherwise, this method generates an error.

Note that this method can be used if the receiver disallows subclasses and has no unnamed instance variables, even if it is not modifiable. Note also that the `instVar:constrainTo:` method cannot be used under those conditions.

`allowSubclasses` Allows creation of subclasses of a class.

`disallowSubclasses` Disallows creation of subclasses of a class. If the receiver is not modifiable, this method generates an error. If the receiver is modifiable and already has subclasses, this method generates an error.

`immediateInvariant` Recompiles all methods for the receiver. If the receiver has a Subclasses class variable, also recompiles all methods for subclasses. This recompilation is performed to check for references to deleted instance variables.

If no errors found during compilation, then makes the receiver immediately invariant.

If no errors found during compilation, and if subclasses are allowed, then makes the receiver's array of **constraints**, and the receiver's array of instance variable names immediately invariant. If the receiver has a Subclasses class variable, it is removed.

If errors occur during compilation, then the receiver is not made invariant.

`instVar:` *aString* `constrainTo:` *aClass*

Changes the receiver's constraint on the instance variable named *aString* to *aClass*.

The argument *aString* must be the name of an instance variable defined in the receiver or inherited from a superclass. *aClass* must be a kind of Class. The receiver, and any subclasses for which a constraint change will result, must be modifiable; otherwise, an error will be generated.

If the superclass of the receiver has a constraint on the same instance variable, then *aClass* must be identical to, or a subclass of, that inherited constraint.

For each of the receiver's subclasses, if the constraint on the specified instance variable is *aClass* or is a subclass of *aClass*, then that constraint will be unchanged. Otherwise, the subclass's constraint will be changed to *aClass*.

`isModifiable` Returns true if the receiver may be modified (that is, if the receiver, its array of **constraints**, and its array of instance variable names are all variant, and the receiver has a subclasses class variable). Returns false otherwise.

`recompileAllMethodsInContext:` *aSymbolList*

Recompiles all methods for the receiver, using the specified symbol list.

This method is designed to allow a user interface to issue GciContinue after fixing the source code for a method in error. GciContinue will reattempt the compilation of the method which contained an error, then proceed to the next method.

`recompileAllSubclassMethodsInContext:` *aSymbolList*

Recompiles all methods for the receiver and its subclasses, using the specified symbol list. If the receiver is not modifiable, then methods in subclasses will not be recompiled, since only modifiable classes should have the Subclasses class variable present.

`removeInstVar:` *aString* Removes the instance variable named *aString* from the receiver and from all of the receiver's subclasses. The receiver and all of its subclasses must be modifiable.

All instance methods for the receiver and its subclasses are recompiled using the symbol list of the current user. If an error occurs during recompilation of methods, the instance variable will have been removed from the receiver and from all of its subclasses, but some methods in some subclasses will not have been recompiled.

You may not use this method to remove an inherited instance variable.

`validateIsModifiable` Returns the receiver if the receiver, its array of **constraints**, and its array of instance variables are modifiable. Generates an error if the receiver cannot be modified (that is, if the receiver, its array of **constraints**, or its array of instance variable names is not variant).

`validateSubclassesAreModifiable`

Generates an error if the receiver or any of its subclasses cannot be modified.

`varyingConstraint:` *aClass*

Changes the constraint on the unnamed variables of the receiver.

The argument *aClass* must be a kind of Class. The receiver, and any subclasses for which a constraint change will result, must be modifiable. Otherwise, an error will be generated.

If the superclass of the receiver has a constraint on its unnamed part, then *aClass* must be identical to, or a subclass of, that inherited constraint.

For each of the receiver's subclasses, if the constraint on that subclass's unnamed part is either *aClass* or a subclass of *aClass*, that constraint will be unchanged. Otherwise, the subclass's constraint will be changed to *aClass*.

Queries

`isSpecial` Returns whether instances of this class have their state encoded in their identities.

Testing

`isBehavior` Returns true if the receiver is a kind of Behavior, and returns false otherwise.

Testing Inheritance

`isSubclassOf:` *aClassHistory*

Returns true if the receiver is identical to or is a subclass of any class in *aClassHistory*; otherwise, returns false.

If the *aClassHistory* argument is actually a class rather than a class history, then this method uses the class history of the argument, instead of the class itself.

`validateSubclassOf:` *aClass*

Returns true if receiver is identical to *aClass* or is a subclass of *aClass*; otherwise, generates an error.

Updating Categories

`addCategory:` *aString* Adds *aString* as a method category for the receiver. If *aString* is already a method category, generates an error.

`moveMethod:` *aSelector* `toCategory:` *categoryName*

Moves the method *aSelector* (a String) from its current category to the specified category (also a String). If either *aSelector* or *categoryName* is not in the receiver's method dictionary, or if *aSelector* is already in *categoryName*, generates an error.

`removeCategory:` *categoryName*

Removes the specified category and all its methods from the receiver's method dictionary. If *categoryName* is not in the receiver's method dictionary, generates an error.

`renameCategory:` *categoryName* `to:` *newCategoryName*

Changes the name of the specified category to *newCategoryName* (a String), and returns the receiver. If *categoryName* is not in the receiver's method dictionary, or if *newCategoryName* is already in the receiver's method dictionary, generates an error.

`renameOrMergeCategory:` *oldName* `to:` *newName*

Changes the name of the specified category to *newName* (a String), and returns the receiver. If *oldName* is not in the receiver's method dictionary, generates an error. If *newName* is already in the receiver's category list, moves all the methods from the old category to the new category, and removes the old category.

Updating the Method Dictionary

`compileAccessingMethodsFor:` *anArrayOfSymbols*

This method is a simple way to create methods for reading and modifying instance variables in instances of the receiver. Each element of *anArrayOfSymbols* must be an instance variable in the receiver. For each instance variable x in the Array, two methods are created: `x` (read the variable) and `x:`*newValue* (modify the variable). The first method (`x`) is placed in the category Accessing, while the second method (`x:`*newValue*) is placed in the category Updating.

The method can also be used to create methods for accessing and modifying class and pool variables. When creating class methods, the message must be sent to the class of the class.

Returns the receiver. Generates an error if any element of *anArrayOfSymbols* is not an instance variable, class variable, or pool variable of the receiver.

`compileMethod:` *sourceString* `dictionaries:` *aSymbolList*
`category:` *aCategoryString*

This compiles some source code for the receiver. The first argument, *sourceString*, is the string of source code to be compiled. The second argument is a SymbolList to be used in parsing, along with the list of all class variables and pool dictionaries for the receiver and all of its superclasses. The third argument (a String) indicates the method's category.

sourceString must be a kind of String or DoubleByteString. Instances of JapaneseString are not supported as source strings. String literals (abc) are generated as instances of the class of *sourceString*, unless *sourceString* is a Symbol, in which case abc produces a String. If *sourceString* is a DoubleByteSymbol, abc produces a DoubleByteString.

If there are no errors, this adds the resulting compiled method to the receiver's method dictionary and returns nil.

If errors occur, the result is an Array of error descriptors which can be be used as an input to the GsMethod (C) | _sourceWithErrors:fromString: method.

An error descriptor is an Array of size 3 or 4, containing the following elements:

1. The GemStone error number.
2. Offset into the source string where the error occurred.
3. Error message text, if available, or nil.
4. Internal compiler error text, if the error is internal.

`removeAllMethods` Removes all methods from the receiver. This should not be done without considerable forethought!

`removeSelector:` *aString*

Removes the method whose selector is *aString* from the receiver's method dictionary. If the selector is not in the method dictionary, generates an error.

Class Protocol

Instance Creation

`new`	Disallowed. To create a new Class or Metaclass, use `Class | subclass:instVarNames:...` instead.
`new:` *anInteger*	Disallowed. To create a new Class or Metaclass, use `Class | subclass:instVarNames:...` instead.

BinaryFloat

BinaryFloat is an abstract class. Various subclasses provide different implementations of Binary floating point. Each subclass is expected to conform to IEEE Standard 754.

Superclasses Number, Magnitude, Object

Named Instance Variables None

Instance Format Pointer, Nonindexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`at:` *anIndex* `put:` *aValue* Disallowed. You may not change the value of a Float.

`denominator` Returns the denominator of a Fraction representing the receiver.

`numerator` Returns the numerator of a Fraction representing the receiver.

`size:` *anInteger* Disallowed. You may not change the size of a Float.

Arithmetic

`abs` Returns a Number that is the absolute value of the receiver.

`factorial` Returns the factorial of the integer part of the receiver. Returns 1 if the receiver is less than or equal to 1.

`negated` Returns a Number that is the negation of the receiver.

`raisedToInteger:` *aNumber*
Returns the receiver raised to the power of the argument.

`rem:` *aNumber* Returns the integer remainder defined in terms of quo: (division of the receiver by *aNumber*, with truncation toward zero).

Comparing

`>=` *aMagnitude* — Returns true if the receiver is greater than or equal to *aMagnitude*; returns false otherwise.

Converting

`asFraction` — Returns a Fraction that represents the receiver. If the receiver is a NaN, or Infinity, returns the receiver.

`asScaledDecimal:` *scale*

Returns a ScaledDecimal that represents the receiver. If the receiver is a NaN or Infinity, returns the receiver. The argument *scale* should be a non-negative SmallInteger.

Formatting

`asString` — (Subclass responsibility.) Returns a String corresponding to the value of the receiver. Where applicable, returns one of the following Strings: PlusInfinity, MinusInfinity, PlusQuietNaN, MinusQuietNaN, PlusSignalingNaN, or MinusSignalingNaN.

`asStringUsingFormat:` *anArray*

(Subclass responsibility.) Returns a String corresponding to the receiver, using the format specified by *anArray*.

Storing and Loading

`writeTo:` *passiveObj* — Converts the receiver to its passive form and writes that information on *passiveObj*.

Testing

`even` — Returns true if the receiver is an even integer, false otherwise.

`negative` — Returns true if the receiver is less than zero, false if the receiver is zero or greater.

`odd` — Returns true if the receiver is an odd integer, false otherwise.

`positive` — Returns true if the receiver is greater than or equal to zero, false if the receiver is less than zero.

`strictlyPositive` — Returns true if the receiver is greater than zero and false if it is less than or equal to zero.

Truncation and Rounding

`fractionPart` Returns the fraction remaining after the receiver is truncated toward zero.

`integerPart` Returns an integer representing the receiver truncated toward zero.

Class Protocol

Arithmetic

`pi` Returns the value of pi, accurate to twenty decimal places.

Exception Handling

Float status flags, exception handlers, and non-default rounding modes are maintained only for a single GemStone Smalltalk execution and are cleared when a new execution begins.

`clearAllExceptions` Clear all raised exceptions.

`clearException:` *aString*

Clears the raised exception type defined by *aString* (divideByZero, inexactResult, invalidOperation, overflow, underflow). If *aString* is not one of these exception types, an error is generated. Raised exceptions are set by GemStone during floating point operations, and must be explicitly cleared with this method.

`enabledExceptions` Returns a list of all raised exceptions.

`on:` *aString* `do:` *aBlock*

The argument *aString* defines the exception type (divideByZero, inexactResult, invalidOperation, overflow, underflow). If *aString* is not one of these, an error is generated.

The three-argument block *aBlock* is evaluated when the specified exception occurs. The three arguments to *aBlock* are:

1. The category of the exception (always GemStoneError)
2. The number of the exception (always rtErrFltException)
3. an Array containing arguments to the exception, to wit:

1. The type of exception (a Symbol, such as #divideByZero),
2. The selector of the offending operation,
3. The default result that would be returned,
4. The first operand to the operation,
5. The second operand to the operation, if any.

The value that the block returns becomes the result of the floating point operation.

Note that underflow and overflow pass an unusual result to the trap handler if the exception is enabled -- In particular, the correct result is biased by a factor of 10 to the 22500 power to bring it into the representable range of a Float.

If you do not want to field the exception specified by *aString*, leave *aBlock* nil. If *aBlock* is neither a block nor nil, an error is generated. Returns the receiver.

`operationException:` *aString*

Returns true if the specified exception has occurred in the current operation. Otherwise, returns false. The argument *aString* defines the exception type (divideByZero, inexactResult, invalidOperation, overflow, underflow). If *aString* is not one of these, an error is generated.

`operationExceptions` Returns a list of all exceptions raised by the last floating point operation.

`raisedException:` *aString*
Returns true if the specified exception has occurred since the last clearException operation. Otherwise, returns false. The argument *aString* defines the exception type (divideByZero, inexactResult, invalidOperation, overflow, underflow). If *aString* is not one of these, an error is generated.

The occurrence of a floating point exception that is not trapped by `on:do:` causes that exception to be raised.

`raisedExceptions` Returns a list of all raised exceptions.

`status` Returns an empty Array in this release.

`status:` *aString* Has no effect in this release.

`trapEnabled:` *aString* Returns true if a trap handler has been defined for the specified exception Otherwise, returns false.

Instance Creation

`fromStream:` *aStream* Generates a BinaryFloat from *aStream*. Generates an error if an attempt is made to read beyond the end of the stream.

The Stream must contain a legal BinaryFloat, as defined by the following BNF construction:

```
BinaryFloat =
    ( Integer '.' Digit {Digit}
      [ E Integer ] ) |
    ( Integer E Integer )
Integer = [ ('+' | '-') ] Digit {Digit}
E = ( 'E' | 'e')
```

Note that the syntax does not allow certain valid BinaryFloats (such as PlusInfinity and MinusInfinity) to be read.

`fromString:` *aString* Returns an instance of Float, constructed from *aString*. The String must contain only characters representing the object to be created, although leading and trailing blanks are permitted.

Storing and Loading

`loadFrom:` *passiveObj*	Reads from *passiveObj* the passive form of an object. Converts the object to its active form by loading the information into a new instance of the receiver. Returns the new instance.

Truncation and Rounding

`roundingMode`	Returns the current rounding mode (nearestEven, towardMinusInfinity, towardPlusInfinity, towardZero). Returns unknown if access to rounding mode is not implemented for the receiver.
`roundingMode:` *aString*	The argument *aString* defines the rounding mode (nearestEven, towardMinusInfinity, towardPlusInfinity, towardZero). If *aString* is not one of these, an error is generated.

BlockClosure

BlockClosure is an abstract superclass for all the different kinds of blocks of GemStone Smalltalk code.

Superclasses Object

Named Instance Variables None

Instance Format Pointer, Nonindexable, Variant

Subclass Creation Disallowed

Instance Protocol

Copying

`copy` Disallowed.

Storing and Loading

`writeTo:` *aPassiveObject* Instances of BlockClosure cannot be converted to passive form. This method writes nil to *aPassiveObject* and stops GemStone Smalltalk execution with a notifier.

Testing

`isSimple` Returns the default answer, false.

Updating

`instVarAt:` *anIndex* `put:` *aValue*
Disallowed.

Class Protocol

Instance Creation

`new` Disallowed.

Boolean

The only two instances of Boolean represent the two logical truth values: true and false.

You may not create new instances of Boolean. You also may not create subclasses of Boolean.

Superclasses Object

Named Instance Variables None

Instance Format Special, Nonindexable, Invariant

Subclass Creation Disallowed

Instance Protocol

Clustering

`clusterDepthFirst` Returns true. (Because Booleans are self-defining objects, this method has no effect.)

Copying

`copy` Overrides the inherited method to return the receiver. The pseudo-variables true and false are the only instances of Boolean, and must preserve identity.

Flow of Control

`and:` *aBlock* (Reserved selector.) Nonevaluating conjunction. Returns the value of the zero-argument block *aBlock* if the receiver is true. Otherwise, returns false without evaluating the argument.

`ifFalse:` *aBlock* (Reserved selector.) Returns the value of the zero-argument block *aBlock* if the receiver is false. Otherwise, returns nil without evaluating the argument.

`ifFalse:` *falseBlock* `ifTrue:` *trueBlock*
(Reserved selector.) Returns the value of the zero-argument block *falseBlock* if the receiver is false. Otherwise, returns the value of the zero-argument block *trueBlock* without evaluating *falseBlock*.

`ifTrue:` *aBlock* (Reserved selector.) Returns the value of the zero-argument block *aBlock* if the receiver is true. Otherwise, returns nil without evaluating the argument.

`ifTrue:` *trueBlock* `ifFalse:` *falseBlock* (Reserved selector.) Returns the value of the zero-argument block *falseBlock* if the receiver is false. Otherwise, returns the value of the zero-argument block *trueBlock* without evaluating *falseBlock*.

`or:` *aBlock* (Reserved selector.) Nonevaluating disjunction. Returns the value of the zero-argument block *aBlock* if the receiver is false. Otherwise, returns true without evaluating the argument.

Formatting

`asString` Returns a String containing true or false, depending on the receiver.

`printString` Returns a String whose contents are a displayable representation of the receiver.

Logical Operations

`&` *aBoolean* Evaluating conjunction (AND). Returns true if both the receiver and the argument *aBoolean* are true.

`eqv:` *aBoolean* Returns true if the receiver is identical to *aBoolean*.

`not` Negation. Returns true if the receiver is false. Returns false if the receiver is true.

`xor:` *aBoolean* Exclusive OR. Returns true if the receiver is not identical to *aBoolean*.

`|` *aBoolean* Evaluating disjunction (OR). Returns true if either the receiver or the argument *aBoolean* is true.

Storing and Loading

`writeTo:` *passiveObj* Converts the receiver to its passive form and writes that information on *passiveObj*.

Class Protocol

Instance Creation

`fromStream:` *aStream*	If the next characters in *aStream* are true or false (case insensitive, leading spaces permitted), this method returns the appropriate Boolean. Otherwise, generates an error.
`fromString:` *aString*	If *aString* contains true or false, returns the appropriate Boolean. Leading and trailing spaces are permitted in the String. If *aString* contains any characters other than true or false, this method generates an error.
`new`	Disallowed. You cannot create new instances of Boolean.

Storing and Loading

`loadFrom:` *passiveObj*	Reads from *passiveObj* the passive form of an object. Converts the object to its active form and returns an equivalent instance of Boolean.

BtreeReadStream

BtreeReadStream supports the composition of query results by providing access to a btree structure.

A BtreeReadStream can read all the entries of the btree one at a time. Supply the root node when you create the stream, and send the next message to read the first node. Send the next message repeatedly to iterate over the btree's contents, and send the atEnd message to check if there are any more nodes.

Superclasses Stream, Object

Named Instance Variables

endNode — The leaf node that is the last entry in the btree.

endIndex — The index of the leaf node that is the last entry in the btree.

currentStack — An Array of btree node / offset pairs. The successive elements of the Array indicate the path through the btree to the current entry.

Instance Format Pointer, Nonindexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`currentStack`	Returns the value of the instance variable **currentStack**.
`endIndex`	Returns the value of the instance variable **endIndex**.
`endNode`	Returns the value of the instance variable **endNode**.
`next`	Returns the next value on a stream of Btree values. Update the current stack for a subsequent `next`.
`size`	Returns the number of elements contained in the receiver (that is, how many successful `next` operations can be performed).

Adding

`nextPut:` *anObject*	Disallowed. You cannot write to a BtreeReadStream.

Testing

`atEnd`	Returns true if the receiver is positioned at the end of the stream, and false otherwise.

Updating

`currentStack:` *newValue*	Modifies the value of the instance variable **currentStack**.
`endIndex:` *newValue*	Modify the value of the instance variable **endIndex**.
`endNode:` *newValue*	Modify the value of the instance variable **endNode**.

Class Protocol

Instance Creation

`on:` *aBtreeNode*	Create a stream that can access the entire contents of the btree whose root node is BtreeNode.

ByteArray

A ByteArray is a SequenceableCollection whose elements are SmallIntegers with a value between zero and 255 inclusive. Uninitialized ByteArray elements are zero.

Superclasses SequenceableCollection, Collection, Object

Named Instance Variables None

Instance Format Byte, Indexable, Variant

Subclass Creation Allowed

Instance Protocol

Comparing

`hash` Returns a positive SmallInteger based on the contents of the receiver.

Converting

`asHexString` Returns a String containing a hexadecimal printed representation of the contents of the receiver. For example, the message `'abc' asHexString` returns the String '616163'.

The receiver must be a byte format object.

CanonicalStringDictionary

A CanonicalStringDictionary is a StringKeyValueDictionary that provides protocol that is similar to Set in addition to its dictionary protocol.

Superclasses StringKeyValueDictionary, KeyValueDictionary, AbstractDictionary, Collection, Object

Named Instance Variables None

Instance Format Pointer, Indexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`includes:` *aString* Returns true if the receiver contains *aString* as a key, false otherwise.

`includesValue:` *aString* Returns true if the receiver contains *aString* as a key, false otherwise.

Hashing

`hashFunction:` *aKey* The hash function performs an operation on the value of the key (*aKey*) and returns a value in the range 1..tableSize.

Updating

`add:` *aString* — Adds *aString* if it is not already present in the receiver, and returns either *aString* or the canonical string already present.

`addAll:` *aCollection* — Adds elements of *aCollection* to the receiver. Returns *aCollection*.

`addAssociation:` *anAssociation* — Add the argument *anAssociation* to the receiver.

`remove:` *aString* — Removes *aString* if present in the receiver and returns the removed value. If *aString* is not present, generates an error.

`remove:` *aString* `ifAbsent:` *aBlock* — Removes *aString* if present in the receiver and returns the removed value. If *aString* is not present, returns the result of evaluating the zero argument Block *aBlock*.

Character

There are 256 characters. You may not create new instances or subclasses of class Character.

Superclasses AbstractCharacter, Magnitude, Object

Named Instance Variables None

Instance Format Special, Nonindexable, Invariant

Subclass Creation Disallowed

Instance Protocol

Accessing

`asciiValue` Returns the ASCII code of the receiver (a SmallInteger).

Case-Insensitive Comparisons

`equalsNoCase:` *aCharacter*

Returns true if the receiver is the same character as the argument regardless of case or internal representation.

`isEquivalent:` *aCharacter*

Returns true if the receiver is the same character as the argument regardless of case or internal representation.

Comparisons

< *aCharacter* Returns true if the ASCII code of the receiver is less than that of *aCharacter*.

<= *aCharacter* Returns true if the ASCII code of the receiver is less than or equal to the ASCII code of *aCharacter*.

= *aCharacter* Returns true if the receiver and *aCharacter* are the same ASCII character.

> *aCharacter* Returns true if the ASCII code of the receiver is greater than the ASCII code of *aCharacter*.

>= *aCharacter* Returns true if the ASCII code of the receiver is greater than or equal to the ASCII code of *aCharacter*.

Converting

`asCharacter`	Returns the receiver.
`asDigit`	Returns the digit value (0-9) of the receiver. If the receiver is not a digit, this returns 0.
`asInteger`	Returns the ASCII value of the receiver.
`asJISCharacter`	Returns the JISCharacter corresponding to the receiver.
`asLowercase`	Returns a Character that is the lowercase character corresponding to the receiver. If the receiver is lowercase or has no case, this returns the receiver itself.
`asSymbol`	Returns a one character Symbol that represents the receiver.
`asUppercase`	Returns a Character that is the uppercase character corresponding to the receiver. If the receiver is uppercase or has no case, this returns the receiver itself.
`digitValue`	Returns a SmallInteger representing the value of the receiver, a digit, or returns nil if the receiver is not a digit.
`digitValueInRadix:` *radix*	Returns a SmallInteger representing the value of the receiver, a digit, or returns nil if the receiver is not a digit in the given radix.

Copying

`copy`	Returns the receiver. (Does not create a new Character.)

Formatting

`asString`	Returns a one-character String or DoubleByteString containing the receiver.
`displayWidth`	Returns the width necessary to display the receiver. For a Character, this method always returns 1.
`printOn:` *aStream*	Puts a displayable representation of the receiver on the given stream.
`printString`	Returns a String whose contents are a displayable representation of the receiver.

Storing and Loading

`writeTo:` *passiveObj*	Converts the receiver to its passive form and writes that information on *passiveObj*.

Testing

`isAlphaNumeric`	Returns true if the receiver is a Roman letter or digit. Returns false otherwise.
`isDigit`	Returns true if the receiver is a digit. Returns false otherwise.
`isLetter`	Returns true if the receiver is a Roman letter. Returns false otherwise.
`isLowercase`	Returns true if the receiver is a lowercase character. Returns false otherwise.
`isSeparator`	Returns true if the receiver is a separator character (space, tab, cr, lf, or newPage). Returns false otherwise.
`isUppercase`	Returns true if the receiver is an uppercase character. Returns false otherwise.
`isVowel`	Returns true if the receiver is a vowel (Y is considered to be a vowel). Returns false otherwise.

Class Protocol

Instance Creation

`fromStream:` *aStream*	Returns the next Character in the stream *aStream*.
`fromString:` *aString*	If *aString* is a one character String, returns the Character in *aString*. Otherwise, generates an error.
`new`	Disallowed. You may not create new instances of Character.
`withValue:` *anInteger*	Returns the Character with the specified value. Allowable range is 0 <= *anInteger* <= 65535.

Non-Printable Characters

`backspace`	Returns the ASCII backspace character.
`cr`	Returns the ASCII carriage return character.
`esc`	Returns the ASCII escape character.
`lf`	Returns the ASCII linefeed character.
`newPage`	Returns the ASCII new page character.
`space`	Returns the ASCII space character.
`tab`	Returns the ASCII tab character.

Printable Characters

`digits`	Returns an InvariantArray containing ASCII characters representing digits 0 through 9.
`lowercaseRoman`	Returns an InvariantArray containing all lowercase Roman ASCII characters in alphabetic order.
`uppercaseRoman`	Returns an InvariantArray containing all uppercase Roman ASCII characters in alphabetic order.

CharacterCollection

CharacterCollection is an abstract superclass for behavior that is common to all indexed collections of Characters.

Subclasses must reimplement the following selectors:

```
at:
at:put:
insertAll:at:
removeFrom:to:
size
size:
```

However these selectors do not generate the subclass-responsibility error (error 2008) because to do so would break the `Object | printString` method.

Superclasses SequenceableCollection, Collection, Object

Named Instance Variables None

Instance Format Pointer, Indexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`byteAt:` *index*
: Considers the receiver as an array of bytes and returns the byte at position *index*.

Adding

`add:` *aCharOrCharColl*
: Appends all of the elements of *aCharOrCharColl* to the receiver and returns *aCharOrCharColl*.

`addAll:` *aCharOrCharCollection*
: Equivalent to `add:` *aCharOrCharCollection*.

`addLast:` *aCharOrCharCollection*
: Equivalent to `add:` *aCharOrCharCollection*.

`insertAll:` *aCharOrCharCollection* `at:` *anIndex*
: Inserts *aCharOrCharCollection* into the receiver at the specified index and returns *aCharOrCharCollection*.

Backward Compatibility

Methods in this category are obsolete and are provided only for compatibility with earlier releases of GemStone. They will be removed in a future release.

`insert:` *aCharOrCharCollection* `at:` *anIndex*

Obsolete in GemStone 5.0. Use the `insertAll:at:` method instead.

`toServerTextFile:` *aFileSpec*

Obsolete in GemStone 4.1. Use an instance of GsFile to access the file system of the client or server machines.

Case-Insensitive Searching

`findStringNoCase:` *subString* `startingAt:` *startIndex*

If a receiver contains *subString* beginning at some point at or after *startIndex*, this returns the index at which *subString* begins. If the receiver does not contain *subString*, this returns 0.

The search is case-insensitive.

`includesString:` *aString*

Returns true if *aString* is contained as a subString within the receiver, using a case-insensitive search. Returns false otherwise.

Case-Sensitive Searching

`findString:` *subString* `startingAt:` *startIndex*

If a receiver contains *subString* beginning at some point at or after *startIndex*, this returns the index at which *subString* begins. If the receiver does not contain *subString*, this returns 0.

The search is case-sensitive.

Comparing

Some of these methods determine whether one String collates before another. In collation, the values of the receiver and aCharCollection are compared character-by-character, from left to right, in case-sensitive fashion. If two CharacterCollections are of different length, and all characters in the shorter collection are equal to their counterparts in the longer one, the shorter collection collates before the longer.

Unlike the comparison methods for the superclass SequenceableCollection, these methods merely require that both the receiver and argument be kinds of CharacterCollection (rather than requiring both to be of the same class).

`<` *aCharCollection* — Returns true if the receiver collates before the argument. Returns false otherwise.

`<=` *aCharCollection* — Returns true if the receiver collates before the argument or if all of the corresponding characters in the receiver and argument are equal. Returns false otherwise.

`=` *aCharCollection* — Returns true if all of the corresponding characters in the receiver and argument are equal. Returns false otherwise.

`>` *aCharCollection* — Returns true if the receiver collates after the argument. Returns false otherwise.

`>=` *aCharCollection* — Returns true if the receiver collates after the argument or if all of the corresponding characters in the receiver and argument are equal. Returns false otherwise.

`at:` *anIndex* `equals:` *aCharCollection*

Returns true if *aCharCollection* is contained in the receiver starting at *anIndex*. Returns false otherwise.

Note that this method returns true only if *aCharCollection* begins exactly at the position designated by *anIndex*. To locate a pattern beginning on or after *anIndex*, see the method `findPattern:startingAt:` in category Searching.

`match:` *prefix* — Returns true if the argument *prefix* is a prefix of the receiver, and false if not. The comparison is case-sensitive.

`matchesAnyOf:` *aCollectionOfCharacterColls*

Returns true if the receiver returns true to the message `match:` with any of the objects in the given collection; returns false otherwise. Examples:

```
xyz matchesAnyOf: #(xyz 'abc*')
 true
xyz matchesAnyOf: #(ayz abc)
 false
x#z matchesAnyOf: #(x@z '*')
 false
x#z matchesAnyOf: #($*)
 true
```

The class JISString does not support this method.

`matchPattern:` *aPattern* Returns true if the receiver matches *aPattern*, false if it doesn't. An exact match is required. For partial matching, use the Searching method `findPattern:startingAt:` instead.

The argument *aPattern* is a kind of Array containing zero or more CharacterCollections, plus zero or more occurrences of the special characters $* or $?. If either $* or $? occurs in *aPattern*, it acts as a wild card. The character $? matches any single character in the receiver, and $* matches any sequence of zero or more characters in the receiver. For example,

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

returns true, because the character $* is interpreted as a wild card.

If either of these special characters occurs in the receiver, it is interpreted literally. For example,

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

returns false - because the character $* occurs in the receiver, it is interpreted as a literal asterisk (not as a wild card).

Concatenating

`,` *aCharOrCharCollection*	Returns a new instance of the receiver's class that contains the elements of the receiver followed by the elements of *aCharOrCharCollection*. Warning: Creating a new instance and copying the receiver take time. If you can safely modify the receiver, it can be much faster to use the `addAll:` method. See the documentation of the Concatenating category of class SequenceableCollection for more details.

Converting

`asArrayOfKeywords`	Returns an Array of keyword substrings held by the receiver. The receiver is assumed to be a colon-separated list of substrings. These substrings are extracted and collected in an Array. If the receiver contains no colons, the Array will hold a copy of the receiver.
`asArrayOfPathTerms`	Returns an Array of path substrings held by the receiver. The receiver is assumed to be a period-separated list of substrings. These substrings are extracted and collected in an Array. If the receiver contains no periods, the Array will hold a copy of the receiver. Periods not meant to separate path terms may be escaped with a `$\` character.
`asArrayOfSubstrings`	Returns an Array of substrings held by the receiver. The receiver is assumed to be a separator-separated list of substrings. These substrings are extracted and collected in an Array. If the receiver contains no separators, the Array will hold a copy of the receiver. Separators not meant to separate substrings may be escaped with a `$\` character.
`asDecimalFloat`	Returns a DecimalFloat whose value is represented by the receiver.
`asDoubleByteString`	Returns a DoubleByteString representation of the receiver.
`asFloat`	Returns a Float whose value is represented by the receiver.

`asHexString`	Returns a String containing a hexadecimal printed representation of the contents of the receiver. For example, the message `'abc' asHexString` returns the String '616163'. The receiver must be a byte format object.
`asInteger`	Returns an Integer whose value is represented by the receiver.
`asJISString`	Returns a JISString representation of the receiver.
`asLowercase`	Returns a new instance of the receiver's class, with all uppercase characters in the receiver changed to lowercase.
`asNumber`	Returns the Integer whose value corresponds to the receiver. The receiver must consist only of an optional minus sign (-) followed by any number of digit characters, $0 through $9. Invalid characters, and any valid characters that follow an invalid character, are ignored.
`asSmallFloat`	Returns a SmallFloat whose value is represented by the receiver.
`asString`	Returns a String representation of the receiver.
`asSymbolKind`	Returns a canonical symbol containing the same characters as the receiver.
`asUppercase`	Returns a new instance of the receiver's class, with all lowercase characters in the receiver changed to uppercase.
`subStrings`	Returns an Array of CharacterCollections where element represents a word in the receiver. A word is a group of Characters separated by one or more separators.
`subStrings:` *aCharacter*	Returns an Array of CharacterCollections in which each element represents a substring separated by *aCharacter*.
`trimBlanks`	Returns a CharacterCollection containing the same Characters as the receiver, but with leading and trailing blanks removed.
`trimLeadingBlanks`	Returns a CharacterCollection containing the same Characters as the receiver, but with leading blanks removed.

`trimLeadingSeparators`

Returns a CharacterCollection containing the same Characters as the receiver, but with leading separators removed.

`trimSeparators` Returns a CharacterCollection containing the same Characters as the receiver, but with leading and trailing separators removed.

`trimTrailingBlanks` Returns a CharacterCollection containing the same Characters as the receiver, but with trailing blanks removed.

`trimTrailingSeparators`

Returns a CharacterCollection containing the same Characters as the receiver, but with trailing separators removed.

Copying

`copyFrom:` *startIndex* `to:` *stopIndex*

Returns a new SequenceableCollection containing the elements of the receiver between *startIndex* and *stopIndex*, inclusive. The result is of the same class as the receiver, unless the receiver is a Symbol or DoubleByteSymbol, in which case the result class is respectively String or DoubleByteString.

Both *startIndex* and *stopIndex* must be positive integers not larger than the size of the receiver, with *startIndex* <= *stopIndex*.

`copyWithout:` *anObject* Returns a copy of the receiver that does not contain the given object. Comparisons are by equality.

Formatting

`describeClassName`	CharacterCollections may be said to describe themselves, so this method returns the receiver.
`printOn:` *aStream*	Puts a displayable representation of the receiver on the given stream.
`quoted`	Returns a copy of the receiver enclosed in single-quote marks, with contained single-quote characters doubled. The copy is of the same class as the receiver.
`width:` *anInteger*	Pads the receiver with spaces to create an object of size *anInteger*. If *anInteger* is positive, the spaces are added to the right of the receiver. If *anInteger* is negative, the spaces are added to the left of the receiver. If the size of the receiver is already greater than *anInteger*, the receiver is left unchanged.
`wrapTo:` *col*	Word-wrap the given character collection to the given column, treating tab characters as modulo-8.

Hashing

`hash`	Returns a positive Integer based on a case-sensitive hash of the contents of the receiver. The algorithm implemented is described in: [Pearson 90] Pearson, Peter K., Fast Hashing of Variable-Length Text Strings, Communications of the ACM 33, 6, (June 1990), 677-680.

Searching

`findPattern:` *aPattern* `startingAt:` *anIndex*

This method searches the receiver, beginning at *anIndex*, for a substring that matches *aPattern*. If a matching substring is found, this method returns the index of the first character of the substring. Otherwise, this returns 0.

The argument *aPattern* is an Array containing zero or more CharacterCollections plus zero or more occurrences of the special characters asterisk or questionMark. See the description of the Comparing method `matchPattern:` for more information about this argument.

Performs a case-sensitive search.

`findPatternNoCase:` *aPattern* `startingAt:` *anIndex*

This method searches the receiver, beginning at *anIndex*, for a substring that matches *aPattern*. If a matching substring is found, this method returns the index of the first character of the substring. Otherwise, this returns 0.

The argument *aPattern* is an Array containing zero or more CharacterCollections plus zero or more occurrences of the special characters asterisk or questionMark. See the description of the Comparing method `matchPattern:` for more information about this argument.

Performs a case-insensitive search.

`indexOf:` *pattern* `matchCase:` *flag* `startingAt:` *startIndex*

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. Otherwise, returns 0.

The argument *pattern* is an Array containing zero or more CharacterCollections plus zero or more occurrences of the special characters asterisk or questionMark. See the description of the Comparing method `matchPattern:` for more information about this argument.

If matchCase is true, a case-sensitive search is performed. Otherwise, a case-insensitive search is performed.

`maxConsecutiveSubstring`

Returns the largest substring within the receiver that contains characters with consecutive ASCII values. For example, the message

```
'abxabcdxabc' maxConsecutiveSubstring
```

yields the result `'abcd'`.

If there are no such substrings larger than 2 characters, returns a String that contains the first character in the receiver.

`maxRepeatingSubstring`

Returns the largest substring within the receiver that contains repetitions of a character, using case-sensitive comparison. For example, the message

```
'aaxbbbBxccc' maxRepeatingSubstring
```

yields the result `'bbb'`.

If there are no such substrings larger than 1 character, returns a String that contains the first character in the receiver.

`maxSameTypeSubstring` Returns the largest substring within the receiver that contains either all digits, all alphabetic characters, or all special characters. For example, the message

```
'axv2435,.-' maxSameTypeSubstring
```

yields the result `'2435'`.

If there are no such substrings larger than 1 character, returns a String that contains the first character in the receiver.

This method may generate an error if the receiver is a JapaneseString.

Testing

`isEquivalent:` *aCharCollection*

Returns true if the receiver is equivalent to *aCharCollection*. The receiver is equivalent to *aCharCollection* if the receiver contains the same characters as *aCharCollection* regardless of case or internal representation. For example, if $a is in *aCharCollection*, it is equivalent to any representation of an a in the receiver's character set.

`sameAs:` *aCharCollection* Returns true if the receiver is equivalent to *aCharCollection*. The receiver is equivalent to *aCharCollection* if the receiver contains the same characters as *aCharCollection* regardless of case or internal representation. For example, if $a is in *aCharCollection*, it is equivalent to any representation of an a in the receiver's character set.

Updating

`addLineDelimiters` Returns a copy of the receiver that contains each occurence of the backslash character replaced by the line-feed character.

`byteAt:` *index* `put:` *newElement*

Considers the receiver as an array of bytes and sets the byte at position *index* to *newElement*. The argument *newElement* replaces any previously stored value and must be an Integer between 0 and 255.

`lf` Appends a line-feed to the receiver and returns the receiver.

`space` Appends a space to the receiver and returns the receiver.

`tab` Appends a tab to the receiver and returns the receiver.

Class Protocol

Backward Compatibility

Methods in this category are obsolete and are provided only for compatibility with earlier releases of GemStone. They will be removed in a future release.

`fromServerTextFile:` *aFileSpec*

Obsolete in GemStone 4.1. Use an instance of GsFile to access the file system of the client or server machines.

Instance Creation

`fromStream:` *aStream* `width:` *anInteger*

Returns a new instance of the receiver's class that contains the next *anInteger* characters of *aStream*.

`withAll:` *aSequenceableCollection*

Returns a new instance of the receiver that contains the elements in the argument *aSequenceableCollection*.

`withBytes:` *aByteObject* Returns a new instance of the receiver that contains the bytes in the argument *aByteObject*.

Class

Each of the classes in the GemStone kernel inherits some of its behavior from Class.

You may send the messages described here to any of the kernel classes (class-defining objects) defined in this manual. However, you may not send these messages to instances of the kernel classes (that is, unless the receiver is an instance of Class).

Consider the following example. The description of class SmallInteger contains two kinds of protocol: instance methods and class methods. Instance methods are understood by SmallIntegers (instances of the class SmallInteger, which inherit their protocol from Integer, Number, Magnitude, and Object). Class methods are understood by the class-defining object SmallInteger itself (which is the single instance of the Metaclass "SmallInteger class", and inherits its protocol from Class, Behavior, and Object). The messages described here (for Class) are understood by SmallInteger; that is, they are class methods for the class-defining object), but are not understood by instances of SmallInteger.

Superclasses

Behavior, Object

Named Instance Variables

name — The class's name for itself; a Symbol of up to 64 Characters.

classHistory — The ClassHistory to which the class belongs. Every class belongs to exactly one class history, which tracks its ancestry and assists with changes to its structure (schema). When a new class is created, it is considered to be either a new version of an existing class, or else it has no previous history. A new class version becomes the most recent version in an existing ClassHistory. Otherwise, a new ClassHistory is created for the new class.

description — Any object, usually an instance of GsClassDocumentation, that describes the class. It can be modified with the `description:` message.

migrationDestination — A Class, generally considered to be the next later version of this class. At an appropriate time, it may be desirable or necessary to migrate instances of this class to the newer version. This variable remembers which class the instance should migrate to.

You can mark a Class with a migration destination by sending it the message `migrateTo:`. When so marked, instances of that Class can be migrated to the new Class while maintaining identity. The destination Class should have the method `migrateFrom:` implemented to define the transformation. A default implementation is provided in Object.

Migration is triggered manually by sending the message `migrate` to an instance of the Class. Other protocol for forcing migration is `Class | migrateInstancesTo:` and `Repository | migrateInstancesOfClasses:`.

timeStamp — A DateTime object that indicates when the class was created.

userId — A CharacterCollection that gives the identity of the user that created the class.

extraDict — Reserved for internal use by GemStone Systems, Inc.

classCategory — A CharacterCollection that names the category of classes to which this class belongs. Each subclass also belongs to this category, unless the subclass overrides it with its own category. Class categorization can be used by browsers and schema design tools.

subclasses — An IdentitySet of the subclasses of this class. This set is only present in modifiable classes, and is nil otherwise.

Instance Format Pointer, Nonindexable, Variant

Subclass Creation Disallowed

Instance Protocol

Accessing

`classHistory`	Returns the **classHistory** instance variable for this class.
`description`	Returns the **description** instance variable of this class.
`extraDict`	Returns the SymbolDictionary held in **extraDict** that holds miscellaneous information about the receiver.
`migrationDestination`	Returns the **migrationDestination** instance variable of this class.
`name`	Returns the receiver's name (the contents of the **name** instance variable).
`timeStamp`	Returns the timestamp instance variable of this class, a DateTime showing when the class was created.
`userId`	Returns the **userId** instance variable of this class, the id of the user who created this class.

Authorization

`changeToSegment:` *seg*	Assigns the receiver and its non-shared components to the given segment. The segments of class variable values are not changed. The current user must have write access to both the old and new segments for this method to succeed.

Backward Compatibility

Methods in this category are obsolete and are provided only for compatibility with earlier releases of GemStone. They will be removed in a future release.

`assignClassToSegment:` *seg*

Obsolete in GemStone 4.0.

`byteSubclass:` *aString* `classVars:` *anArrayOfClassVars*
`classInstVars:` *anArrayOfClassInstVars*
`poolDictionaries:` *anArrayOfPoolDicts* `inDictionary:` *aDictionary*
`description:` *aDescription* `isInvariant:` *invarBoolean*

Obsolete in GemStone 4.1. The preferred methods are in the Subclass Creation category. Look for the method that omits this method's keyword `description:` and changes its keyword `isInvariant:` to `instancesInvariant:`.

`byteSubclass:` *aString* `classVars:` *anArrayOfClassVars*
`classInstVars:` *anArrayOfClassInstVars*
`poolDictionaries:` *anArrayOfPoolDicts* `inDictionary:` *aDictionary*
`inClassHistory:` *aClassHistory* `description:` *aDescription*
`isInvariant:` *invarBoolean*

Obsolete in GemStone 4.0. The preferred methods are in the Subclass Creation category.

`byteSubclass:` *aString* `classVars:` *anArrayOfClassVars*
`classInstVars:` *anArrayOfClassInstVars*
`poolDictionaries:` *anArrayOfPoolDicts* `inDictionary:` *aDictionary*
`newVersionOf:` *oldClass* `description:` *aDescription*
`isInvariant:` *invarBoolean*

Obsolete in GemStone 4.1. The preferred methods are in the Subclass Creation category. Look for the method that omits this method's keyword `description:` and changes its keyword `isInvariant:` to `instancesInvariant:`.

`byteSubclass:` *aString* `classVars:` *anArrayOfClassVars*
`poolDictionaries:` *anArrayOfPoolDicts* `inDictionary:` *aDictionary*
`inClassHistory:` *aClassHistory* `description:` *aDescription*
`isInvariant:` *invarBoolean*

Obsolete in GemStone 4.0. The preferred methods are in the Subclass Creation category.

`byteSubclass:` *aString* `classVars:` *anArrayOfClassVars*
`poolDictionaries:` *anArrayOfPoolDicts* `inDictionary:` *aDictionary*
`isInvariant:` *invarBoolean*

Obsolete in GemStone 4.0. The preferred methods are in the Subclass Creation category.

`indexableSubclass:` *aString* `instVarNames:` *anArrayOfStrings*
`classVars:` *anArrayOfClassVars* `classInstVars:` *anArrayOfClassInstVars*
`poolDictionaries:` *anArrayOfPoolDicts* `inDictionary:` *aDictionary*
`constraints:` *aConstraint* `instancesInvariant:` *invarBoolean*
`description:` *aDescription* `isModifiable:` *modifyBoolean*

Obsolete in GemStone 4.1. The preferred methods are in the Subclass Creation category. Look for the method that omits this method's keyword `description:`.

`indexableSubclass:` *aString* `instVarNames:` *anArrayOfStrings*
`classVars:` *anArrayOfClassVars* `classInstVars:` *anArrayOfClassInstVars*
`poolDictionaries:` *anArrayOfPoolDicts* `inDictionary:` *aDictionary*
`constraints:` *aConstraint* `instancesInvariant:` *invarBoolean*
`inClassHistory:` *aClassHistory* `description:` *aDescription*
`isModifiable:` *modifyBoolean*

Obsolete in GemStone 4.0. The preferred methods are in the Subclass Creation category.

`indexableSubclass:` *aString* `instVarNames:` *anArrayOfStrings*
`classVars:` *anArrayOfClassVars* `classInstVars:` *anArrayOfClassInstVars*
`poolDictionaries:` *anArrayOfPoolDicts* `inDictionary:` *aDictionary*
`constraints:` *aConstraint* `instancesInvariant:` *invarBoolean*
`newVersionOf:` *oldClass* `description:` *aDescription*
`isModifiable:` *modifyBoolean*

Obsolete in GemStone 4.1. The preferred methods are in the Subclass Creation category. Look for the method that omits this method's keyword `description:`.

`indexableSubclass:` *aString* `instVarNames:` *anArrayOfStrings*
`classVars:` *anArrayOfClassVars* `poolDictionaries:` *anArrayOfPoolDicts*
`inDictionary:` *aDictionary* `constraints:` *aConstraint*
`instancesInvariant:` *invarBoolean* `inClassHistory:` *aClassHistory*
`description:` *aDescription* `isModifiable:` *modifyBoolean*

Obsolete in GemStone 4.0. The preferred methods are in the Subclass Creation category.

`indexableSubclass:` *aString* `instVarNames:` *anArrayOfStrings*
`classVars:` *anArrayOfClassVars* `poolDictionaries:` *anArrayOfPoolDicts*
`inDictionary:` *aDictionary* `constraints:` *aConstraint*
`instancesInvariant:` *invarBoolean* `isModifiable:` *modifyBoolean*

Obsolete in GemStone 4.0. The preferred methods are in the Subclass Creation category.

`indexableSubclass:` *aString* `instVarNames:` *anArrayOfStrings*
`classVars:` *anArrayOfClassVars* `poolDictionaries:` *anArrayOfPoolDicts*
`inDictionary:` *aDictionary* `constraints:` *aConstraint*
`isInvariant:` *invarBoolean*

Obsolete in GemStone 4.0. The preferred methods are in the Subclass Creation category.

`subclass:` *aString* `inDictionary:` *aDictionary* `constraints:` *constraintSpec*

Obsolete in GemStone 4.0. The preferred methods are in the Subclass Creation category.

`subclass:` *aString* `instVarNames:` *anArrayOfStrings*
`classInstVars:` *anArrayOfClassInstVars* `inDictionary:` *aDictionary*
`isModifiable:` *modifyBoolean*

Obsolete in GemStone 4.0. The preferred methods are in the Subclass Creation category.

`subclass:` *aString* `instVarNames:` *anArrayOfStrings*
`classVars:` *anArrayOfClassVars* `classInstVars:` *anArrayOfClassInstVars*
`poolDictionaries:` *anArrayOfPoolDicts* `inDictionary:` *aDictionary*
`constraints:` *aConstraint* `instancesInvariant:` *invarBoolean*
`description:` *aDescription* `isModifiable:` *modifyBoolean*

Obsolete in GemStone 4.1. The preferred methods are in the Subclass Creation category. Look for the method that omits this method's keyword `description:`.

`subclass:` *aString* `instVarNames:` *anArrayOfStrings*
`classVars:` *anArrayOfClassVars* `classInstVars:` *anArrayOfClassInstVars*
`poolDictionaries:` *anArrayOfPoolDicts* `inDictionary:` *aDictionary*
`constraints:` *aConstraint* `instancesInvariant:` *invarBoolean*
`inClassHistory:` *aClassHistory* `description:` *aDescription*
`isModifiable:` *modifyBoolean*

Obsolete in GemStone 4.0. The preferred methods are in the Subclass Creation category.

`subclass:` *aString* `instVarNames:` *anArrayOfStrings*
`classVars:` *anArrayOfClassVars* `classInstVars:` *anArrayOfClassInstVars*
`poolDictionaries:` *anArrayOfPoolDicts* `inDictionary:` *aDictionary*
`constraints:` *aConstraint* `instancesInvariant:` *invarBoolean*
`newVersionOf:` *oldClass* `description:` *aDescription*
`isModifiable:` *modifyBoolean*

Obsolete in GemStone 4.1. The preferred methods are in the Subclass Creation category. Look for the method that omits this method's keyword `description:`.

`subclass:` *aString* `instVarNames:` *anArrayOfStrings*
`classVars:` *anArrayOfClassVars* `poolDictionaries:` *anArrayOfPoolDicts*
`inDictionary:` *aDictionary* `constraints:` *aConstraint*
`instancesInvariant:` *invarBoolean* `inClassHistory:` *aClassHistory*
`description:` *aDescription* `isModifiable:` *modifyBoolean*

Obsolete in GemStone 4.0. The preferred methods are in the Subclass Creation category.

`subclass:` *aString* `instVarNames:` *anArrayOfStrings*
`classVars:` *anArrayOfClassVars* `poolDictionaries:` *anArrayOfPoolDicts*
`inDictionary:` *aDictionary* `constraints:` *aConstraint*
`instancesInvariant:` *invarBoolean* `isModifiable:` *modifyBoolean*

Obsolete in GemStone 4.0. The preferred methods are in the Subclass Creation category.

`subclass:` *aString* `instVarNames:` *anArrayOfStrings*
`classVars:` *anArrayOfClassVars* `poolDictionaries:` *anArrayOfPoolDicts*
`inDictionary:` *aDictionary* `constraints:` *aConstraint*
`isInvariant:` *invarBoolean*

Obsolete in GemStone 4.0. The preferred methods are in the Subclass Creation category.

`subclass:` *aString* `instVarNames:` *anArrayOfStrings*
`inDictionary:` *aDictionary* `isModifiable:` *modifyBoolean*

Obsolete in GemStone 4.0. The preferred methods are in the Subclass Creation category.

Browser Methods

`changeNameTo:` *newInternalName*
Sets the receiver's **name** instance variable.

`compileMissingAccessingMethods`
Creates accessing and updating methods for all instance variables that do not already have such methods.

`definition` Returns a String containing a GemStone Smalltalk definition for the receiver (that is, a subclass creation message). This method uses the UserProfile of the owner of the current session as the correct context.

`hierarchy` Returns a String that enumerates the receiver's superclasses (up to Object) and the instance variables defined by the receiver and each of its superclasses.

`recompileWithDicts:` *symbolList*
Recompiles all the receiver's instance and class methods. Returns the compiledMethods that fail to compile properly.

Category

`category:` *newCategory* Sets the **classCategory** variable of the receiver. The argument should be a kind of CharacterCollection or nil.

Class Instance Variables

`addClassInstanceVariable:` *civNameString*

Adds the given class instance variable to the receiver's metaclass. Generates an error if the receiver is either not modifiable or does not disallow subclasses.

`atClassInstVar:` *varName*

Returns the value of the given class instance variable in the receiver. Generates an error if the argument does not name a class instance variable in the receiver. In general, it is more efficient to implement a direct accessing method for a class instance variable.

`atClassInstVar:` *varName* `put:` *newValue*

Changes the value of the given class instance variable in the receiver, without regard to the variance or invariance of the receiver. Generates an error if the argument does not name a class instance variable in the receiver. Returns the argument *newValue*.

Clustering

`clusterBehavior` Clusters elements of the receiver and its metaclass that are used for GemStone Smalltalk execution.

It is recommended that when several classes are being clustered in a transaction, send `clusterBehavior` to all classes to be clustered, then send `clusterDescription`.

`clusterDepthFirst` Clusters elements of the receiver and its metaclass that are used for GemStone Smalltalk execution, then Clusters elements of the receiver and its metaclass that are not required for GemStone Smalltalk execution.

`clusterDescription` Clusters elements of the receiver and its metaclass that are not required for GemStone Smalltalk execution.

It is recommended that when several classes are being clustered in a transaction, send `clusterBehavior` to all classes to be clustered, then send `clusterDescription`.

Decompiling without Sources

`decompileMethods:` *selectorsToDecompile* `classRefExpression:` *refString* `stripWith:` *stripSelector* `includeAll:` *includeAll*

Returns a String that contains topaz commands to regenerate methods for the receiver.

If *selectorsToDecompile* is nil, all methods will be decompiled, otherwise *selectorsToDecompile* should be a Collection of Symbols and only those methods listed in *selectorsToDecompile* will be included.

If *includeAll* is true, all methods not decompiled will be filed out in source form and included in the result.

stripSelector should be the selector of an instance method in CompiledMethod to be used in stripping the source strings. Examples are #emptySource , #sourceToFirstComment, #fullSource .

refString is a String containing an expression which evaluates to the class. If *refString* is nil, the **name** of the receiver is used.

Displaying

`instanceString` Returns a symbol that can be used to name an instance of the receiver.

`instanceSymbol` Returns a symbol that can be used to name an instance of the receiver.

Instance Migration

`allInstances` Returns an IdentitySet that contains all instances of the receiver.

Note: This method scans the entire GemStone repository, and may therefore take some time to execute.

Note: This method returns all instances of the receiver that have not already been reclaimed, even instances that are not connected to the repository and have been marked as candidates for garbage collection.

`cancelMigration` Disables class migration by clearing the **migrationDestination** instance variable.

`instVarMappingTo:` *anotherClass*

Returns an instance-variable mapping from the receiver's named instance variables to those in the given class. If an entry is 0, the other class does not have the corresponding instance variable.

`migrateInstances:` *instances* `to:` *anotherClass*

Migrates each of the instances to *anotherClass*, using `migrateFrom:instVarMap:` and performing become: operations to accomplish this task. Removes the indexes of indexed instances. Returns an Array of four Sets of instances, none of which were migrated:

- Objects that you cannot read.
- Objects that you cannot write.
- Objects that are in indexed collections that have different formats. (For a more detailed description, see `Object | become:`.)
- Objects whose class is not identical to the receiver.

Generates the error errNotSameClassHist if the **classHistory** of the receiver is not identical to the **classHistory** of *anotherClass*.

`migrateInstancesTo:` *anotherClass*

Finds all instances of the receiver. Migrates each instance that is accessible and whose references are writable to *anotherClass*, using migrateFrom:instVarMap: and performing become: operations to accomplish this task. Removes the indexes of indexed instances. Returns an Array of four Sets of instances, none of which were migrated:

- Objects that you cannot read.
- Objects that you cannot write.
- Objects that are in indexed collections that have different formats. (For a more detailed description, see `Object | become:`.)
- Objects whose class is not identical to the receiver.

This method scans the entire GemStone repository, and may therefore take some time to execute.

`migrateTo:` *aClass*

Enables class migration by setting the **migrationDestination** instance variable.

Locking

`lockableParts`

Returns an array of the receiver's contents that are locked by browsers and folders.

Queries

`isMeta`

Returns whether the receiver is a kind of MetaClass.

`thisClass`

Returns this class. This class's MetaClass returns this class as well. This method is useful to get the base version of a class if one is holding either the class or its metaclass.

Subclass Creation

Every new GemStone Smalltalk class must be a subclass of some other existing GemStone Smalltalk class. To create the new class, you send a subclass creation message to its intended superclass.

The following restrictions apply to creating classes:

- The new class must be of the same implementation (storage format) as the receiver (its superclass), unless the receiver is a non-indexable pointer object. In this case, there are no restrictions if the receiver has no instance variables. If the receiver does have instance variables, the new class may not be of special or byte format.
- The name of a class is a Symbol at most 64 characters long.
- The name of an instance variable is a String at most 64 characters long.
- A class contains at most 255 named instance variables.

Implementation Format. Instance variables may be named or unnamed. The class definition (often in the subclass creation method) explicitly declares the name and number of all named instance variables. This definition must be fixed (class not modifiable) before instances of the class can be created. The class definition also implicitly declares unnamed instance variables (if they exist), by the choice of implementation format. Unnamed variables can vary in number independently for each instance. Depending upon format, unnamed variables may be indexed (in which case they are accessed by index), or not (in which case they are unordered and are accessed associatively, by value). Classes in byte format have indexed instance variables that are stored by byte for efficiency of storage and access.

You use different methods to create a byte class, an indexable class, or a class of another format. For each of these possibilities there is a pair of standard methods. Each of these methods provides a full (long) list of keywords that permit you to specify a new class fully. One of them also allows you explicitly to specify the new class as a version of an existing class, while the other does not. Additional methods provide selected shorter lists of keywords for convenience, and supply default values for some arguments.

Pool dictionaries. If you want to add or remove pool dictionaries for the new class at some later time, the argument that supplies the Array of pool dictionaries must not be an array literal. The literal value produces an InvariantArray object, which cannot subsequently be modified.

Dictionary. GemStone adds the new class to a dictionary. The dictionary is typically already in the current user's symbol list, but it can be added to the symbol list at a later time if it is not already there. (The symbol list makes the class visible to the user.) The specified dictionary is often UserGlobals, but may be Globals if the data curator has authorized the user to modify that dictionary.

Constraints. Constraints restrict the ranges of values for instance variables. When a constrained instance variable is assigned a value, GemStone Smalltalk ensures that the value either is nil or is an object whose class is of a given kind. It raises an error if the constraint is not satisfied.

An individual constraint is specified as a two-element array. The first element is a Symbol that gives the name of an instance variable, and the second element is the class to which that instance variable is constrained. A constraint is allowed to name any instance variable that is available to the new class, whether it is defined directly in that class or in one of its superclasses. A new constraint for a variable defined in a superclass must be at least as restrictive as any constraint that applies to the superclass.

The argument that specifies constraints is a literal Array of individual constraints (that is, an Array of two-element Arrays). However, if the new class is indexable or it is a non-sequenceable collection, the argument array may optionally contain a final element that is a Class (rather than a two-element Array). This class specifies a constraint on all of the new class's unnamed instance variables.

In the following example, the unnamed variables of the new subclass SubAssembly are constrained to contain instances of Part:

```
Array subclass: #SubAssembly
instVarNames: #(#name, #partNum) "the parts list is unnamed"
...
constraints: #[ #[partNum, Integer],
              "unnamed variables: the parts list" Part ]
...
```

Invariance. The *invarBoolean* argument of a subclass creation method deals with class-level invariance. When that argument is true, GemStone thereafter forces all instances of the new class to become invariant as soon as they are committed to GemStone. That is, invariance applies to all objects of that class.

If instances of the new class's superclass are invariant, then instances of the new class must also be invariant. In this case, a subclass creation method generates an error if the invarBoolean argument is not true.

Class Modification. The *modifyBoolean* argument of a subclass creation method deals with object-level invariance, the ability to modify the object that is the class itself.

Classes are typically not modifiable. As a result, this argument is generally given the value false. The subclass creation method then makes the new class an invariant object, and instances of that class can be created at any time after.

When the *modifyBoolean* argument is true, the new class is modifiable, not invariant. Its constraints and instance variables can be modified. However, no instances of it can yet be created. Once all desired changes have been made, you must send the new class the message `immediateInvariant`. That message then makes the new class an invariant object, and no further changes to it are possible. However, instances of the class can then be created.

For more information about invariance at all levels, see the *GemStone Programming Guide*.

Classes and Schema. A class can be viewed as an implementation of a schema, or of part of a schema. In order to define and develop a schema, you may create modifiable classes, which remain modifiable until the schema is stable.

However, it is sometimes also necessary to change schema after classes are no longer modifiable, and after instances of them exist. To accomplish this kind of change, you must create new classes to implement the new schema. However, it may be desirable to consider a new class to be a new version of an existing class, so that a logical connection between them and their instances can be maintained.

Speaking conceptually, a class history lists all the versions of a class. Speaking technically, the objects that are classes do not have versions. Versions are represented by the list of classes in a class history. Every class (object) belongs to exactly one class history; therefore, all the classes that are listed in a class history share the same class history object.

Subclass methods that have an oldClass argument typically create the new class as a new version of oldClass, and the two classes then share the same class history. However, if oldClass is nil, then the new class is no relation to any existing class, and it has a new class history.

When subclass methods that lack the *oldClass* argument create a new class with the same name as another class that is visible to the user, then the new class is a new version of the existing class, and they share the same class history. However, if no existing class of this name is visible to the user, then the new class is no relation to any existing class, and it has a new class history.

`byteSubclass:` *aString* `classVars:` *anArrayOfClassVars*
`classInstVars:` *anArrayOfClassInstVars*
`poolDictionaries:` *anArrayOfPoolDicts* `inDictionary:` *aDictionary*
`instancesInvariant:` *invarBoolean*

Creates and returns a new byte subclass of the receiver. You are not permitted to modify the new class after it is created. If the receiver is not some kind of String class, then instances of the new class store and return SmallIntegers in the range 0 - 255.

If *aString* is the name of a Class that is visible to the current user, this method creates the new class as a new version of the existing class, and they then share the same class history. However, if no class named *aString* is visible to the user, then the new class is no relation to any existing class, and it has a new class history.

This method generates an error if instances of the receiver are of special storage format, if they are NSCs, or if they have instance variables.

`byteSubclass:` *aString* `classVars:` *anArrayOfClassVars*
`classInstVars:` *anArrayOfClassInstVars*
`poolDictionaries:` *anArrayOfPoolDicts* `inDictionary:` *aDictionary*
`newVersionOf:` *oldClass* `instancesInvariant:` *invarBoolean*

Creates and returns a new byte subclass of the receiver. You are not permitted to modify the new class after it is created. If the receiver is not some kind of String class, then instances of the new class store and return SmallIntegers in the range 0 - 255.

If *oldClass* is visible to the current user, this method creates the new class as a new version of *oldClass*, and the two classes then share the same class history. However, if *oldClass* is nil, then the new class is no relation to any existing class, and it has a new class history.

This method generates an error if instances of the receiver are of special storage format, if they are NSCs, or if they have instance variables.

`indexableSubclass:` *aString* `instVarNames:` *anArrayOfStrings*
`classVars:` *anArrayOfClassVars* `classInstVars:` *anArrayOfClassInstVars*
`poolDictionaries:` *anArrayOfPoolDicts* `inDictionary:` *aDictionary*
`constraints:` *aConstraint* `instancesInvariant:` *invarBoolean*
`isModifiable:` *modifyBoolean*

Creates and returns a new indexable subclass of the receiver. Instances of the new class are represented as pointer objects.

If *aString* is the name of a Class that is visible to the current user, this method creates the new class as a new version of the existing class, and they then share the same class history. However, if no class named *aString* is visible to the user, then the new class is no relation to any existing class, and it has a new class history.

This method generates an error if instances of the receiver are of special storage format or if they are NSCs.

`indexableSubclass:` *aString* `instVarNames:` *anArrayOfStrings*
`classVars:` *anArrayOfClassVars* `classInstVars:` *anArrayOfClassInstVars*
`poolDictionaries:` *anArrayOfPoolDicts* `inDictionary:` *aDictionary*
`constraints:` *aConstraint* `instancesInvariant:` *invarBoolean*
`newVersionOf:` *oldClass* `isModifiable:` *modifyBoolean*

Creates and returns a new indexable subclass of the receiver. Instances of the new class are represented as pointer objects.

If *oldClass* is visible to the current user, this method creates the new class as a new version of *oldClass*, and the two classes then share the same class history. However, if *oldClass* is nil, then the new class is no relation to any existing class, and it has a new class history.

This method generates an error if instances of the receiver are of special storage format or if they are NSCs.

`subclass:` *aString* `instVarNames:` *anArrayOfStrings*
`classVars:` *anArrayOfClassVars* `classInstVars:` *anArrayOfClassInstVars*
`poolDictionaries:` *anArrayOfPoolDicts* `inDictionary:` *aDictionary*
`constraints:` *aConstraint* `instancesInvariant:` *invarBoolean*
`isModifiable:` *modifyBoolean*

Creates and returns a new subclass of the receiver.

If *aString* is the name of a Class that is visible to the current user, this method creates the new class as a new version of the existing class, and they then share the same class history. However, if no class named *aString* is visible to the user, then the new class is no relation to any existing class, and it has a new class history.

`subclass:` *aString* `instVarNames:` *anArrayOfStrings*
`classVars:` *anArrayOfClassVars* `classInstVars:` *anArrayOfClassInstVars*
`poolDictionaries:` *anArrayOfPoolDicts* `inDictionary:` *aDictionary*
`constraints:` *aConstraint* `instancesInvariant:` *invarBoolean*
`newVersionOf:` *oldClass* `isModifiable:` *modifyBoolean*

Creates and returns a new subclass of the receiver.

If *oldClass* is visible to the current user, this method creates the new class as a new version of *oldClass*, and the two classes then share the same class history. However, if *oldClass* is nil, then the new class is no relation to any existing class, and it has a new class history.

`subclass:` *aString* `instVarNames:` *anArrayOfStrings*
`inDictionary:` *aDictionary*

Creates and returns a new subclass of the receiver.

This method is a shortcut for convenience only. It may not be retained in future GemStone releases. Use it interactively or pedagogically, but avoid it in production code.

The new class has no class variables, no class instance variables, no pool dictionaries, and no constraints beyond those inherited from the receiver. Instances of the new class are variant, but the new class itself is not modifiable.

If *aString* is the name of a Class that is visible to the current user, this method creates the new class as a new version of the existing class, and they then share the same class history. However, if no class named *aString* is visible to the user, then the new class is no relation to any existing class, and it has a new class history.

`subclass:` *aString* `instVarNames:` *anArrayOfStrings*
`inDictionary:` *aDictionary* `constraints:` *constraintSpec*

Creates and returns a new subclass of the receiver.

This method is a shortcut for convenience only. It may not be retained in future GemStone releases. Use it interactively or pedagogically, but avoid it in production code.

The new class has no class variables, no class instance variables, and no pool dictionaries. Instances of the new class are variant, but the new class itself is not modifiable.

If *aString* is the name of a Class that is visible to the current user, this method creates the new class as a new version of the existing class, and they then share the same class history. However, if no class named *aString* is visible to the user, then the new class is no relation to any existing class, and it has a new class history.

Updating

`addNewVersion:` *aClass* Make *aClass* a new version of the receiver. That is, add *aClass* to the receiver's history, and set aClass's history to be the same as the receiver's history. The existing history of *aClass* will have *aClass* removed from it.

`classHistory:` *aClassHistory*
Sets the value of the **classHistory** instance variable. For use only when creating a class, while the class is not yet invariant.

`description:` *aDescription*
Update the **description** of this Class. Returns the argument.

`extraDict:` *aSymbolDictionary*
Set the value of the **extraDict** instance variable.

`migrationDestination:` *aClass*
Update the **migrationDestination** instance variable. Returns the argument.

`timeStamp:` *aDateTime* Set the value of the **timeStamp** instance variable. For use only when creating a class, while the class is not yet invariant.

`userId:` *aString* Set the value of the **userId** instance variable. For use only when creating a class, while the class is not yet invariant.

Updating Variables

`addClassVarName:` *aString*

Add *aString* to the class variable list for the receiver, if the class variable is not already defined.

`addSharedPool:` *aDictionary*

Add *aDictionary* to the end of the shared pool list for the receiver.

You may use this method only if, when the receiver was created, the argument to `poolDictionaries:` was an Array (rather than a literal Array, which would create an InvariantArray). See `Class | subclass:`.

`removeClassVarName:` *aString*

Remove *aString* from the class variable list for the receiver. Generates an error if *aString* is not specified as a class variable in the receiver.

`removeSharedPool:` *aDictionary*

Remove *aDictionary* from the shared pool list for the receiver. Generates an error if *aDictionary* is not a shared pool for the receiver.

You may use this method only if, when the receiver was created, the argument to `poolDictionaries:` was an Array rather than a literal Array, which would create an InvariantArray. (See `Class | subclass:`.)

Versions

`isVersionOf:` *anotherClass*

Returns whether the receiver and the given class share the same class history.

ClassHistory

A ClassHistory is a sequence of Class objects that logically represent the historical revisions to a Class.

Superclasses Array, SequenceableCollection, Collection, Object

Named Instance Variables

description — A textual description of the function of the Class.

name — The class history's name for itself; a Symbol of up to 64 Characters.

Instance Format Pointer, Indexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`at:` *aTimeOrIndex*

Returns the Class that was current at the given time. The time may be specified absolutely using a DateTime, or relatively using an integer. If a DateTime is specified, returns the version of the class that was active at that time, or nil if the time is before the earliest version.

If an Integer is specified, it is used to chronologically select the version, with 1 indicating the first version created, 2 the version, and so on. If the index is less than one or greater than the number of versions in the history, an error is generated.

`current` Returns the current, or most recent class.

`currentVersion` Returns the most recent version in the receiver's collection of versions.

`description` Returns the **description** of this ClassHistory.

`name` Returns the **name** of this ClassHistory.

Updating

`description:` *aString* Updates the **description** of this ClassHistory.

`name:` *aString* Updates the **name** of this ClassHistory.

`newVersion:` *aClass* Installs the given class as the receiver's most current version. Does not install the receiver in the given class as its version history. Returns the class object.

`removeVersion:` *aClass* Removes the given class from the receiver's list of versions.

Class Protocol

Instance Creation

`new` Create a new ClassHistory.

Updating

`unifyClassHistories:` *anArrayOfClasses*
Creates a new instance of the receiver containing all classes in the argument, and modifies each class in the argument to have the new ClassHistory as its classHistory.

Generates an error and does not modify any class if any element of the argument is not a Class.

ClassOrganizer

A ClassOrganizer collects classes from the current user's symbol list and organizes them into searchable tables that allow tools to present the classes and to perform cross-referencing and fileout.

An organizer can also be created to work with a subtree of another organizer's hierarchy. Such organizers do not track categorization of classes but only the subtree of the overall hierarchy.

Superclasses Object

Named Instance Variables

classes — A ClassSet of all the classes found by an instance.

classNames — Class name information.

user — Reserved for future use.

hierarchy — An IdentityDictionary of class->subclasses associations.

categories — A SymbolDictionary of category->classes associations.

rootClass — The root class of the instance.

Instance Format Pointer, Nonindexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`categories`	Returns the value of the instance variable **categories**.
`classCompletion`	Returns the AutoComplete holding the class names.
`classes`	Returns the ClassSet of classes held by the receiver.
`classNames`	Returns the Array of classnames held by the receiver.
`hierarchy`	Returns the value of the instance variable **hierarchy**.
`rootClass`	Returns the root class for this organizer.

Class Collection

`update` Causes the receiver to rescan for **classes** and rebuild internal structures. Synonymous with `updateClassInfo`.

`updateClassInfo` Causes the receiver to rescan for **classes** and rebuild internal structures.

Fileout Aids

`determineClassFileoutOrder:` *classdict*
Returns an ordered collection of the values that are classes in *classdict*, specifying the order of fileout. The argument should be a SymbolDictionary.

`fileOutClasses:` *order* `on:` *stream* `inDictionary:` *dict* `named:` *dictName*
Writes out code on the given stream that creates the given classes in the dictionary with the given name. The *dict* argument should be a SymbolDictionary of classes.

`fileOutClassesAndMethodsInDictionary:` *aSymbolDictionary* `on:` *aStream*
Files out all source code for **classes** in *aSymbolDictionary* in Topaz filein format on *aStream*.

`fileOutMethods:` *classdict* `order:` *order* `on:` *stream*
File out each class's code and embedded classes.

`fileOutOtherMethods:` *methodInfo* `on:` *stream*
Files out a set of methods on the given stream/file. *methodInfo* must be an array of pairs: #(class selector).

Queries

`allSubclassesOf:` *aClass*
Returns a collection of all the subclasses of the given class: an array that holds a depth-first traversal of the class hierarchy subtree rooted at *aClass*.

`categoryCrossReference`
Returns a dictionary of all method categories and the classes with methods in each category.

`subclassesOf:` *aClass* Returns a copy of the set of subclasses for the given class. Generates an error if the receiver does not hold the given class.

Reporting

`allReferencesTo:` *selector*
: Returns an array of two Arrays. The first contains GsMethods that implement, send, or refer to the given selector. The second contains the indexes into sourceStrings where the first reference takes place.

`allReferencesTo:` *aSelector* `in:` *classSet*
: Returns an array of two Arrays. The first contains GsMethods that implement, send, or refer to the given selector. The second contains the indexes into sourceStrings where the first reference takes place.

`hierarchyReport`
: Returns a String that is a class hierarchy report for all classes known to the receiver.

`implementorsOf:` *aSelector*
: Returns a collection of GsMethods that implement the given selector.

`implementorsOf:` *aSelector* `in:` *aclassSet*
: Returns a collection of GsMethods that implement the given selector.

`referencesTo:` *aSymbol*
: Returns an Array of two sequenceable collections. The first contains GsMethods that refer to the given symbol, and the second contains corresponding indexes into sourceStrings where the first reference takes place.

`referencesTo:` *aSymbol* `in:` *aclassSet*
: Returns an Array of two sequenceable collections. The first contains GsMethods that reference the given symbol as a literal, and the second contains corresponding indexes into sourceStrings where the first such reference takes place.

`searchForCategory:` *catname* `in:` *classSet*
: Returns a collection of GsMethods in the given category.

`sendersOf:` *aSelector*
: Returns an Array of two Arrays. The first subarray contains GsMethods that send the given selector. The second subarray contains indexes where the first use of the selector occurs within the sourceString of the method.

`sendersOf:` *aSelector* `in:` *aclassSet*

Returns an Array of two Arrays. The first subarray contains GsMethods that send the given selector. The second subarray contains indexes where the first use of the selector occurs within the sourceString of the method.

`substringSearch:` *aString*

Returns an Array of two Arrays. The first subarray contains GsMethods whose sources include the given substring. The second subarray contains indexes where the first occurrence of the substring was found.

`substringSearch:` *aString* `in:` *aclassSet*

Returns an Array of two Arrays. The first subarray contains GsMethods whose sources include the given substring. The second subarray contains indexes where the first occurrence of the substring was found.

Reports

`categoryCrossReferenceByName`

Returns a String containing a report from a cross-reference of method categories.

Updating

`addClass:` *cls* — Adds the class *cls*, replacing any existing class with the same superclass.

`classes:` *aClassSet* — Updates the set of **classes** held by the receiver. The receiver's **hierarchy** should be rebuilt after this (see `rebuildHierarchy`).

`recategorize:` *class* `to:` *newCategory*

Move the class from its present category to the given category.

`rootClass:` *aClass* — Sets the root class of the receiver's **hierarchy**. Not generally a useful thing to do.

Class Protocol

Instance Creation

`new`
Creates and returns a new instance of ClassOrganizer with a root of Object.

`newWithRoot:` *aClass* `from:` *anotherOrganizer*
Creates a new ClassOrganizer that is limited to the given subtree of objects.

ClassSet

A ClassSet is an IdentitySet that holds only Class objects.

Superclasses	IdentitySet, IdentityBag, UnorderedCollection, Collection, Object
Named Instance Variables	None
Instance Format	Nsc, Nonindexable, Variant
Subclass Creation	Allowed

Instance Protocol

Sorting

`sortAscending`	Returns an Array with the same (Class) elements as the receiver, in ascending order by class name.

ClusterBucket

A ClusterBucket describes clustering behavior, such as the identity of the extent to which clustered objects are to be written.

Superclasses Object

Named Instance Variables

extentId — A SmallInteger.

keepClusteredOnModify — A Boolean.

description — Unconstrained, typically a String.

Instance Format Pointer, Nonindexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`clusterId` Returns the value of the private instance variable. This instance variable should only be assigned by ClusterBucket | new.

`description` Returns the value of the **description** instance variable.

`extentId` Returns the value of the **extentId** instance variable.

`keepClusteredOnModify`
This feature is not implemented in this release.

Updating

`description:` *anObject*
Assigns *anObject* (typically some kind of String object) as the **description** of the receiver.

`extentId:` *anExtentId*
An argument of nil specifies don't care behavior. The bucket uses the next available disk page from any extent based on the allocation mode defined in the stone's configuration file. Positive arguments are an offset into the result of Repostory | fileNames, thus specifying an extent. Reference to an extent which does not exist will generate an error at the time of executing this method.

At the time of object modification, a non-nil **extentId** specifies which extent to put the object in. If the extent no longer exists at the time of clustering or object modification, don't care behavior occurs.

`keepClusteredOnModify:` *aBoolean*
This feature is deferred until a future release. Argument value of true is not supported in this release.

Class Protocol

Accessing

`allInstances` Returns the collection of all instances of the receiver.

`bucketWithId:` *aPositiveSmallInt*
Returns the instance with the specified id if one exists. Generates an error if anInt is less than 1 or outside of the range of existing cluster buckets.

Accessing the Class Format

`firstPublicInstVar` Returns the index of the first publicly available instance variable storage location, whether or not a public instance variable has actually been defined.

Instance Creation

`new` Creates an instance of the receiver and adds the new instance to the AllClusterBuckets array. Requires write authorization to the DataCurator segment.

`newForExtent:` *extentId* Creates an instance of the receiver for clustering objects in the extent *extentId*. The *extentId* argument is a positive SmallInteger in the range of 1 to (`SystemRepository numberOfExtents`). Requires write authorization to the DataCurator segment.

ClusterBucketArray

A ClusterBucketArray is an Array whose elements are instances of ClusterBucket.

There is one instance of ClusterBucketArray in a fresh GemStone repository. It can be accessed via the Globals dictionary using the following GemStone SmallTalk statement:

```
Globals at: #AllClusterBuckets.
```

GemStone uses AllClusterBuckets to translate clusterIds to cluster bucket objects:

```
classmethod: ClusterBucket
bucketWithId: aSmallPositiveInt
  ^ AllClusterBuckets at: aSmallPositiveInt
```

AllClusterBuckets has a reserved object identifier, to facilitate efficient access of the array from within the object manager. Therefore, a special C constant, OOP_ALL_CLUSTER_BUCKETS, is defined in the `gcioop.ht` header file to permit access to it from C.

Superclasses	Array, SequenceableCollection, Collection, Object
Named Instance Variables	None
Instance Format	Pointer, Indexable, Variant
Subclass Creation	Allowed

Instance Protocol

Clustering

`cluster`
Instances of ClusterBucketArray, especially AllClusterBuckets must always be clustered in the default bucket.

`clusterInBucket:` *aClusterBucketOrId*
Instances of ClusterBucketArray, especially AllClusterBuckets must always be clustered in the default bucket.

`moveToDisk`
Instances of ClusterBucketArray, especially AllClusterBuckets must always be clustered in the default bucket.

`moveToDiskInBucket:` *aClusterBucketOrId*
Instances of ClusterBucketArray, especially AllClusterBuckets must always be clustered in the default bucket.

Collection

Collection is an abstract superclass for all classes whose instances represent a collection of other objects that are known as their elements. It defines methods for operating upon the elements as a whole.

You should not add elements to or remove elements from a Collection at the same time as you are iterating over all or part of the Collection. Doing so may have unpredictable consequences. For example, avoid changing a Collection within the block argument for methods like `do:`, `collect:`, `select:`, `reject:`, and their variants or extensions.

Superclasses Object

Named Instance Variables None

Instance Format Pointer, Indexable, Variant

Subclass Creation Allowed

Instance Protocol

Adding

`add:` *newObject* Makes *newObject* one of the receiver's elements and returns *newObject*.

`addAll:` *aCollection* Adds all of the elements of *aCollection* to the receiver and returns *aCollection*.

Clustering

`clusterDepthFirst` This method clusters the receiver and its named and unnamed instance variables in depth-first order. Returns true if the receiver has already been clustered during the current transaction; returns false otherwise.

Comparing

= *aCollection*	Returns true if all of the following conditions are true: 1. The receiver and *aCollection* are of the same class. 2. The two collections are of the same size. 3. The corresponding elements of the receiver and *aCollection* are equal.
`hash`	Returns a numeric hash key for the receiver.

Converting

`asArray`	Returns an Array with the contents of the receiver.
`asBag`	Returns a Bag with the contents of the receiver.
`asByteArray`	Returns an Array with the contents of the receiver.
`asIdentityBag`	Returns an IdentityBag with the contents of the receiver.
`asIdentitySet`	Returns an IdentitySet with the contents of the receiver.
`asOrderedCollection`	Returns an OrderedCollection with the contents of the receiver.
`asSet`	Returns a Set with the contents of the receiver.
`asSortedCollection`	Returns a SortedCollection with the contents of the receiver.
`asSortedCollection:` *sortBlock*	Returns a SortedCollection with the contents of the receiver, using the given sort block.
`asSortedOrderedCollection`	Returns an OrderedCollection that has been sorted with a SortedCollection and having the contents of the receiver.

Enumerating

`collect:` *aBlock*

Evaluates *aBlock* with each of the receiver's elements as the argument. Collects the resulting values into a collection of the same class as the receiver, and returns the new collection. The argument *aBlock* must be a one-argument block.

For SequenceableCollections, the result preserves the ordering of the receiver. That is, if element a comes before element b in the receiver, then element a is guaranteed to come before b in the result.

`detect:` *aBlock*

Evaluates *aBlock* repeatedly, with elements of the receiver as the argument. Returns the first element for which *aBlock* evaluates to true. If none of the receiver's elements evaluates to true, generates an error. The argument *aBlock* must be a one-argument block.

`detect:` *aBlock* `ifNone:` *exceptionBlock*

Evaluates *aBlock* repeatedly, with elements of the receiver as the argument. Returns the first element for which *aBlock* evaluates to true. If none of the receiver's elements evaluates to true, this evaluates the argument *exceptionBlock* and returns its value. The argument *aBlock* must be a one-argument block, and *exceptionBlock* must be a zero-argument block.

`do:` *aBlock*

Evaluates the one-argument block *aBlock* using each element of the receiver in order. Returns the receiver.

`inject:` *aValue* `into:` *aBlock*

Accumulates a running value associated with evaluating the argument, *aBlock*, with the current value and the each element of the receiver as block arguments. The initial value is the value of the argument, *aValue*. For example:

```
total := #(1 2 3 4) inject: 0
           into: [:sum:int | sum + int]
```

`reject:` *aBlock*
Evaluates *aBlock* with each of the receiver's elements as the argument. Stores the values for which *aBlock* is false into a collection of the same class as the receiver, and returns the new collection. The argument *aBlock* must be a one-argument block.

For SequenceableCollections, the result preserves the ordering of the receiver. That is, if element a comes before element b in the receiver, then element a is guaranteed to come before b in the result.

`select:` *aBlock*
Evaluates *aBlock* with each of the receiver's elements as the argument. Stores the values for which *aBlock* is true into a collection of the same class as the receiver, and returns the new collection. The argument *aBlock* must be a one-argument block.

For SequenceableCollections, the result preserves the ordering of the receiver. That is, if element a comes before element b in the receiver, then element a is guaranteed to come before b in the result.

The new collection that this method returns does not retain any indexes of the receiver. If you want to perform indexed selections on the new collection, you must build all of the necessary indexes. For more information, see the *GemStone Programming Guide*.

`speciesForCollect`
Returns a class, an instance of which should be used as the result of `collect:` or other projections applied to the receiver.

Error Handling

`errorDifferentSizeCollections`
Reports an error indicating that the size of the receiver collection is different from the size of the argument collection.

`errorInvalidArgClass:` *argument* `classes:` *classArray*
Reports an error indicating that the class of *argument* is not one of those specified in *classArray*.

Formatting

`printOn:` *aStream*
Puts a displayable representation of the receiver on the given stream.

Removing

`remove:` *oldObject*
Removes from the receiver an object that is equivalent to *oldObject* and returns *oldObject*. If several elements of the receiver are equivalent to *oldObject*, only one instance is removed. If oldElement has no equivalent elements in the receiver, raises an error.

`remove:` *oldObject* `ifAbsent:` *anExceptionBlock*
Removes from the receiver an object that is equivalent to *oldObject* and returns *oldObject*. If several elements of the receiver are equivalent to *oldObject*, only one instance is removed. If oldElement has no equivalent elements in the receiver, *anExceptionBlock* is evaluated and the result of the evaluation is returned.

`removeAll:` *aCollection*
For each element in *aCollection*, removes from the receiver one element that is equivalent to the element in *aCollection*. Returns *aCollection* if successful.

`removeAllIdentical:` *aCollection*
For each element in *aCollection*, removes from the receiver one element that is identical to the element in *aCollection*. Returns *aCollection* if successful.

`removeIdentical:` *oldObject*
Removes from the receiver an object that is identical to *oldObject*, and returns *oldObject*. If several elements of the receiver are identical to *oldObject*, only one instance is removed. If oldElement is not present in the receiver, raises an error.

`removeIdentical:` *oldObject* `ifAbsent:` *anExceptionBlock*
Removes from the receiver an object that is identical to *oldObject* and returns *oldObject*. If several elements of the receiver are identical to *oldObject*, only one instance is removed. If oldElement is not present in the receiver, *anExceptionBlock* is evaluated and the result of the evaluation is returned.

Searching

`identicalOccurrencesOf:` *anObject*
Returns the number of the receiver's elements that are identical (==) to *anObject*.

`includes:` *anObject* Returns true if *anObject* is equal to one of the elements of the receiver. Returns false otherwise.

`includesIdentical:` *anObject*
Returns true if *anObject* is identical to one of the elements of the receiver. Returns false otherwise.

`includesValue:` *anObject*
Returns true if *anObject* is equal to one of the elements of the receiver. Returns false otherwise.

`occurrencesOf:` *anObject*
Returns the number of the receiver's elements that are equal to *anObject*.

Sorting

`sortAscending` Returns an Array containing the elements of the receiver sorted in ascending order.

`sortAscending:` *aSortSpec*
Returns an Array containing the elements of the receiver, sorted in ascending order, as determined by the values of the instance variables represented by *aSortSpec*. The argument *aSortSpec* must be either a String representing a single path, or an Array holding up to 16 such Strings (each representing a path). If *aSortSpec* is an Array, the first path in the Array is the primary sort key, and the remaining paths are taken in order as subordinate keys.

Each path in *aSortSpec* must follow the rules for equality indexes. In addition, if any path in *aSortSpec* is an empty path (that is, a zero-length String), the sort is performed upon the elements of the receiver itself, rather than upon the instance variables of those elements.

`sortDescending` Returns an Array containing the elements of the receiver sorted in descending order.

`sortDescending:` *aSortSpec*

Returns an Array containing the elements of the receiver, sorted in descending order, as determined by the values of the instance variables represented by *aSortSpec*. The argument *aSortSpec* must be either a String representing a single path, or an Array holding up to 16 such Strings (each representing a path). If *aSortSpec* is an Array, the first path in the Array is the primary sort key, and the remaining paths are taken in order as subordinate keys.

Each path in *aSortSpec* must follow the rules for equality indexes. In addition, if any path in *aSortSpec* is an empty path (that is, a zero-length String), the sort is performed upon the elements of the receiver itself, rather than upon the instance variables of those elements.

`sortWith:` *aSortPairArray*

Returns an Array containing the elements of the receiver, sorted according to the contents of *aSortPairArray*. The argument *aSortPairArray* is an Array of Strings that represent path/direction pairs, in the following form:

```
aCollection sortWith:
   #(a.b ASCENDING a.c DESCENDING ...)
```

That Array may contain up to 16 path/direction pairs. The first path in the Array is the primary sort key, and the remaining paths are taken in order as subordinate keys.

In *aSortPairArray*, each path String must follow the rules for equality indexes. Each direction String must be either ASCENDING or DESCENDING (case-insensitive); otherwise, an error is generated.

In addition, if any path in *aSortPairArray* is an empty path (that is, a zero-length String), the sort is performed upon the elements of the receiver itself, rather than upon the instance variables of those elements.

Testing

`isEmpty` Returns true if the receiver is empty. Returns false otherwise.

`notEmpty` Returns true if the receiver is not empty. Returns false otherwise.

Class Protocol

Instance Creation

`with:` *aValue*
Returns an instance of the receiver containing the argument.

`with:` *aValue* `with:` *val2*
Returns an instance of the receiver containing the arguments.

`with:` *aValue* `with:` *val2* `with:` *val3*
Returns an instance of the receiver containing the arguments.

`with:` *aValue* `with:` *val2* `with:` *val3* `with:` *val4*
Returns an instance of the receiver containing the arguments.

`withAll:` *aCollection*
Returns an instance of the receiver containing the elements of the argument.

CollisionBucket

A CollisionBucket is an Array that is used in a KeyValueDictionary to store a collection of key/value pairs for which the keys hash to the same value.

Superclasses AbstractCollisionBucket, Array, SequenceableCollection, Collection, Object

Named Instance Variables **keyValueDictionary —** An AbstractDictionary. For GemStone internal use.

Instance Format Pointer, Indexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`keyValueDictionary` Returns the value of the instance variable.

Updating

`keyValueDictionary:` *aDict*

Updates the value of the **keyValueDictionary** instance variable.

ComplexBlock

A ComplexBlock references a variable context to access variables in enclosing scopes.

The GemStone virtual machine creates all complex blocks. When a complex block becomes active, it does not require the creation of a variable context because it has no nested blocks that refer to its arguments or temporaries. The virtual machine allocates the block's arguments and temporaries on the execution stack.

Superclasses ExecutableBlock, BlockClosure, Object

Named Instance Variables

selfValue — The value of self for the block; that is, the object that received the message that created this block.

staticLink — A VariableContext that contains the values of the enclosing scope variables that this block may access. This variable can never be nil. If it were, this block would be a SimpleBlock.

Instance Format Pointer, Nonindexable, Variant

Subclass Creation Disallowed

Instance Protocol

Accessing

`selfValue` Returns the value of the instance variable **selfValue**.

`staticLink` Returns the value of the instance variable **staticLink**.

Block Evaluation

`value` Return the value of the receiver evaluated with no arguments. If the block expects any arguments, an error is generated.

`value:` *anObject* Return the value of the receiver evaluated with *anObject* as its argument. If the block expects a different number of arguments, an error is generated.

`value:` *firstObject* `value:` *secondObject*
Return the value of the receiver evaluated with the two objects as its arguments. If the block expects a different number of arguments, an error is generated.

`value:` *firstObject* `value:` *secondObject* `value:` *thirdObject*
Return the value of the receiver evaluated with the three objects as its arguments. If the block expects a different number of arguments, an error is generated.

`value:` *first* `value:` *second* `value:` *third* `value:` *fourth*
Return the value of the receiver evaluated with the four objects as its arguments. If the block expects a different number of arguments, an error is generated.

`value:` *first* `value:` *second* `value:` *third* `value:` *fourth* `value:` *fifth*
Return the value of the receiver evaluated with the five objects as its arguments. If the block expects a different number of arguments, an error is generated.

`valueWithArguments:` *argList*
Return the value of the receiver evaluated with the elements of the Array *argList* as arguments. If the block expects a different number of arguments, an error is generated.

ComplexVCBlock

A ComplexVCBlock is a special kind of complex block that does require the creation of a variable context when it becomes active. It contains a nested block that refers to its arguments or temporaries, which must be allocated in the variable context.

Superclasses	ComplexBlock, ExecutableBlock, BlockClosure, Object
Named Instance Variables	None
Instance Format	Pointer, Nonindexable, Variant
Subclass Creation	Disallowed

Date

An instance of Date describes a date after December 31, 1900.

You can convert a Date to a String (using Formatting instance methods), and you can convert a String to a Date (using Instance Creation class methods). Such conversions require a specification to describe the format of the String. Some methods provide for the default format, DD/MM/YYYY, which expresses the day and month (in that order) as digits.

Explicit string-formatting specifications take the form of an Array, described in Table 2.1. A specification is incorrect if it is missing an element or if an element value is not one of the acceptable values listed in the table.

Table 2.1 String-formatting Specification Array for Date

Element	Acceptable Value	Explanation
1st, 2nd, and 3rd	Integers 1, 2, and 3, in any order	Determines the position of the day (1), month (2), and year (3).
4th	A Character literal (such as a space, $)	Separates year, month, and day.
5th	Integer	Determines the month format to be a number (1), three-letter abbreviation (2), or the entire name (3).
6th	Integer	Determines the year format to be the entire number (1), or only the last two digits (2).

Superclasses

Magnitude, Object

Class Variables

MonthNames — A SymbolDictionary. Each key is a Symbol representing one of the native languages supported by GemStone, and each value is an Array of Strings, the names of the months of the year in the corresponding language.

WeekDayNames — A SymbolDictionary. Each key is a Symbol representing one of the native languages supported by GemStone, and each value is an Array of Strings, the names of the days of the week in the corresponding language.

Named Instance Variables

year — A SmallInteger greater than 1900 and less than 1,000,001 that represents the year.

dayOfYear — A SmallInteger between 1 and 366 inclusive that represents the day of the year.

Instance Format Pointer, Nonindexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`at:` *anIndex* `put:` *aValue*	Disallowed. You may not change the value of a Date.
`dayOfMonth`	Returns a SmallInteger that gives the day of the month described by the receiver.
`dayOfWeek`	Returns a SmallInteger that gives the numeric index of the day of the week described by the receiver. The index is a number between 1 and 7 inclusive, where 1 signifies Sunday.
`dayOfYear`	Returns a SmallInteger that gives the day of the year described by the receiver.
`daysInMonth`	Returns a SmallInteger that gives the number of days in the month described by the receiver.
`daysInYear`	Returns a SmallInteger that gives the number of days in the year described by the receiver.
`julianDay`	Returns the Julian Day of the receiver, a SmallInteger that gives the number of days since January 1, 4713 B.C., as defined in Communications of the ACM, algorithm #199.
`leap`	Returns true if the receiver describes a leap year and false if it does not.
`monthName`	Returns a String that gives the name of the month of the year described by the receiver, in the user's native language.
`monthOfYear`	Returns a SmallInteger that gives the numeric index of the month of the year described by the receiver. The index is a number between 1 and 12 inclusive, where 1 signifies January.

`size:` *anInteger*	Disallowed. You may not change the size of a Date.
`weekDayName`	Returns a String that gives the name of the day of the week described by the receiver, in the user's native language.
`year`	Returns a SmallInteger that gives the year described by the receiver.

Arithmetic

`addDays:` *anInteger*	Returns a Date that describes a date *anInteger* days later than that of the receiver.
`subtractDate:` *aDate*	Returns a positive Integer that counts the number of times midnight occurs between the times described by the receiver and *aDate*.
`subtractDays:` *anInteger*	Returns a Date that describes a date *anInteger* days earlier than that of the receiver.

Comparing

`<` *aDate*	Returns true if the receiver represents a date before that of the argument, and false if it doesn't. Generates an error if the argument is not a Date.
`=` *aDate*	Returns true if the receiver represents the same date as that of the argument, and false if it doesn't.
`>` *aDate*	Returns true if the receiver represents a date after that of the argument, and false if it doesn't. Generates an error if the argument is not a Date.
`hash`	Returns an Integer hash code for the receiver.

Converting

`asDays` — Returns an Integer that represents the receiver in units of days since January 1, 1901, Greenwich Mean Time.

Formatting

`asString` — Returns a String that expresses the receiver in the default format (DD/MM/YYYY).

`asStringUsingFormat:` *anArray*

Returns a String that expresses the receiver in the format defined by *anArray*. Generates an error if *anArray* contains an incorrect formatting specification.

See the class documentation of Date for a complete description of the String-formatting specification Array.

`printOn:` *aStream* — Puts a displayable representation of the receiver, expressed in local time, on *aStream*.

`USDateFormat` — Returns a String that expresses the date of the receiver in local time. The date is in United States format, month first (MM/DD/YY).

Storing and Loading

`writeTo:` *passiveObj* — Writes the passive form of the receiver into *passiveObj*.

Class Protocol

General Inquiries

`nameOfMonth:` *anIndex*
Returns a String that gives the name, in the user's native language, of the month of the year whose numeric index is *anIndex*. The index is a number between 1 and 12 inclusive, where 1 signifies January.

Instance Creation

`fromStream:` *aStream*
Creates and returns an instance of the receiver by reading a String from *aStream*. The String expresses the date in the default format (DD/MM/YYYY). Generates an error if the String does not conform to the format.

`fromStream:` *aStream* `usingFormat:` *anArray*
Creates and returns an instance of the receiver by reading a String from *aStream*. The String expresses the date in the format specified by *anArray*. The expression is terminated either by a space character or by the end of the Stream. Generates an error if the String does not conform to the format, or if *anArray* contains an incorrect formatting specification.

See Table 2.1 for a complete description of the String-formatting specification Array.

If the month format (5th element) indicates either an abbreviation (2) or an entire name (3), then this method tries to determine the month by decoding a character substring. That substring may include any number of characters, but must exactly match a legal month name (or the initial portion of that month name). If the substring matches more than one month, the first month matched is used (the search begins with January).

`fromString:` *aString*
Creates and returns an instance of the receiver from the String *aString*. The String expresses the date in the default format (DD/MM/YYYY). Generates an error if the String does not conform to the format.

`fromString:` *aString* `usingFormat:` *anArray*

Creates and returns an instance of the receiver from the String *aString*. The String expresses the date in the format specified by *anArray*. The expression is terminated either by a space character or by the end of the String. Generates an error if the String does not conform to the format, or if *anArray* contains an incorrect formatting specification.

See Table 2.1 for a complete description of the String-formatting specification Array.

If the month format (5th element) indicates either an abbreviation (2) or an entire name (3), then this method tries to determine the month by decoding a character substring. That substring may include any number of characters, but must exactly match a legal month name (or the initial portion of that month name). If the substring matches more than one month, the first month matched is used (the search begins with January).

`new` Disallowed. To create a new Date, use another instance creation method.

`new:` *anInteger* Disallowed. To create a new Date, use another instance creation method.

`newDay:` *dayInt* `month:` *monthString* `year:` *yearInt*

Creates and returns an instance of the receiver from the specified values. Generates an error if any of the values are out of range.

`newDay:` *day* `monthNumber:` *month* `year:` *year*

Creates and returns an instance of the receiver from the specified values. Generates an error if any of the values are out of range.

`newDay:` *day* `year:` *year*

Creates and returns an instance of the receiver from the specified values. Generates an error if any of the values are out of range.

`today` Creates and returns an instance of the receiver from the system calendar on the machine that is running the Gem process, which is assumed to represent the current date.

Storing and Loading

`loadFrom:` *passiveObj*	Creates and returns an active instance of the receiver from the passive form of the object.

DateTime

An instance of DateTime describes a moment in time (with one-second resolution) on a date after December 31, 1900.

The internal representation of a DateTime is based on Greenwich Mean Time. However, many methods express time in the local timezone. ("Local" time is local to your Gem process.) These methods automatically convert between timezones, but the internal representation remains in Greenwich Mean Time. Hence, you can interact with DateTime methods in a natural way, but DateTime objects can be safely compared to each other no matter what time zone is used to express them.

You can convert a DateTime to a String (using Formatting instance methods), and you can convert a String to a DateTime (using Instance Creation class methods). Such conversions require a specification to describe the format of the String. Some methods provide for the default format, DD/MM/YYYY HH:MM:SS, which expresses the day and month (in that order) as digits and uses a 24-hour clock.

Explicit string-formatting specifications take the form of an Array, described in Table 2.2. A specification is incorrect if it is missing an element or if an element value is not one of the acceptable values listed in the table.

Table 2.2 String-formatting Specification Array for DateTime

Element	Acceptable Value	Explanation
1st, 2nd, and 3rd	Integers 1, 2, and 3, in any order	Determines the position of the day (1), month (2), and year (3).
4th	A Character literal (such as a space, $)	Separates year, month, and day.
5th	Integer	Determines the month format to be a number (1), three-letter abbreviation (2), or the entire name (3).
6th	Integer	Determines the year format to be the entire number (1), or only the last two digits (2).
7th	A Character literal (such as $: or $.)	Separates hours, minutes, and seconds.
8th	true	Include the time of day.
8th	false	Do not include the time of day. Ignore elements 7, 9, and 10. Elements 9 and 10 are optional in the specification.
9th	true	Include seconds.
9th	false	Do not include seconds.
10th	true	Time is expressed in 12-hour format, with am or pm (such as 1:30:55 pm). The space is required preceding the am or pm indicator.
10th	false	Time is expressed in 24-hour format (such as 13:30:55).

Superclasses Date, Magnitude, Object

Named Instance Variables **seconds —** The number of seconds since midnight, Greenwich Mean Time.

Instance Format Pointer, Nonindexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`dayOfMonth`	Returns a SmallInteger that gives the day of the month described by the receiver, expressed in local time.
`dayOfMonthGmt`	Returns a SmallInteger that gives the day of the month described by the receiver, expressed in Greenwich Mean Time.
`dayOfWeek`	Returns a SmallInteger that gives the numeric index of the day of the week described by the receiver, expressed in local time. The index is a number between 1 and 7 inclusive, where 1 signifies Sunday.
`dayOfWeekGmt`	Returns a SmallInteger that gives the numeric index of the day of the week described by the receiver, expressed in Greenwich Mean Time. The index is a number between 1 and 7 inclusive, where 1 signifies Sunday.
`dayOfYear`	Returns a SmallInteger that gives the day of the year described by the receiver, expressed in local time.
`dayOfYearGmt`	Returns a SmallInteger that gives the day of the year described by the receiver, expressed in Greenwich Mean Time.
`daysInMonthGmt`	Returns a SmallInteger that gives the number of days in the month described by the receiver, expressed in Greenwich Mean Time.
`hours`	Returns a SmallInteger (between zero and 23 inclusive) that gives the number of hours represented by the receiver since midnight, local time.
`hoursGmt`	Returns a SmallInteger (between zero and 23 inclusive) that gives the number of hours represented by the receiver since midnight, Greenwich Mean Time.
`leap`	Returns true if the receiver describes a leap year, expressed in local time, and false if it does not.
`leapGmt`	Returns true if the receiver describes a leap year, expressed in Greenwich Mean Time, and false if it does not.

`minutes`	Returns a SmallInteger (between zero and 59 inclusive) that gives the number of minutes represented by the receiver since the previous hour, local time.
`minutesGmt`	Returns a SmallInteger (between zero and 59 inclusive) that gives the number of minutes represented by the receiver since the previous hour, Greenwich Mean Time.
`monthNameGmt`	Returns a String that gives the name of the month of the year described by the receiver, expressed in Greenwich Mean Time, in the user's native language.
`monthOfYear`	Returns a SmallInteger that gives the numeric index of the month of the year described by the receiver, expressed in local time. The index is a number between 1 and 12 inclusive, where 1 signifies January.
`monthOfYearGmt`	Returns a SmallInteger that gives the numeric index of the month of the year described by the receiver, expressed in Greenwich Mean Time. The index is a number between 1 and 12 inclusive, where 1 signifies January.
`seconds`	Returns a SmallInteger (between zero and 59 inclusive) that gives the number of seconds represented by the receiver since the previous minute.
`year`	Returns a SmallInteger that gives the year described by the receiver, expressed in local time.
`yearGmt`	Returns a SmallInteger that gives the year described by the receiver, expressed in Greenwich Mean Time.

Arithmetic

`addDays:` *anInteger*	Returns a DateTime that describes a moment in time *anInteger* days later than that of the receiver.
`addHours:` *aNumber*	Returns a DateTime that describes a moment in time *aNumber* hours later than that of the receiver.
`addMinutes:` *aNumber*	Returns a DateTime that describes a moment in time *aNumber* minutes later than that of the receiver.

`addMonths:` *anInteger*
Returns a DateTime that describes a moment in time *anInteger* months later than that of the receiver.

This method attempts to keep the day of the month the same. If the new month has fewer days than the receiver's original month, then it truncates to the last day of the new month.

`addSeconds:` *aNumber*
Returns a DateTime that describes a moment in time *aNumber* seconds later than that of the receiver.

`addWeeks:` *anInteger*
Returns a DateTime that describes a moment in time *anInteger* weeks later than that of the receiver.

`addYears:` *anInteger*
Returns a DateTime that describes a moment in time *anInteger* years later than that of the receiver.

`subtractDate:` *aDateTime*
Returns a positive Integer that counts the number of times midnight local time occurs between the times described by the receiver and *aDateTime*.

`subtractDateGmt:` *aDateTime*
Returns a positive Integer that counts the number of times that midnight Greenwich Mean Time occurs between the times described by the receiver and *aDateTime*.

`subtractDays:` *anInteger*
Returns a DateTime that describes a moment in time *anInteger* days earlier than that of the receiver.

`subtractHours:` *aNumber*
Returns a DateTime that describes a moment in time *aNumber* hours earlier than that of the receiver.

`subtractMinutes:` *aNumber*
Returns a DateTime that describes a moment in time *aNumber* minutes earlier than that of the receiver.

`subtractMonths:` *anInteger*

Returns a DateTime that describes a moment in time *anInteger* months earlier than that of the receiver.

This method attempts to keep the day of the month the same. If the new month has fewer days than the receiver's original month, then it truncates to the last day of the new month.

`subtractSeconds:` *aNumber*

Returns a DateTime that describes a moment in time *aNumber* seconds earlier than that of the receiver.

`subtractTime:` *aDateTime*

Returns an Array of three positive Integers that count the hours, minutes, and seconds, respectively, between the times of day described by the receiver and *aDateTime*.

The computation ignores the dates of both the receiver and *aDateTime*, and assumes that the receiver is the later time. Hence, if the time of day in the receiver is less than the time of day in *aDateTime*, then the receiver's time of day is assumed to occur on the day following that of *aDateTime*.

`subtractWeeks:` *anInteger*

Returns a DateTime that describes a moment in time *anInteger* weeks earlier than that of the receiver.

`subtractYears:` *anInteger*

Returns a DateTime that describes a moment in time *anInteger* years earlier than that of the receiver.

Backward Compatibility

Methods in this category are obsolete and are provided only for compatibility with earlier releases of GemStone. They will be removed in a future release.

`julianSecond` Obsolete in GemStone 5.0.

Comparing

`<` *aDateTime*	Returns true if the receiver represents a moment in time before that of the argument, and false if it doesn't. Generates an error if the argument is not a DateTime.
`=` *aDateTime*	Returns true if the receiver represents the same moment in time as that of the argument, and false if it doesn't.
`>` *aDateTime*	Returns true if the receiver represents a moment in time after that of the argument, and false if it doesn't. Generates an error if the argument is not a DateTime.
`hash`	Returns an Integer hash code for the receiver.

Converting

`asDateTime`	Returns the receiver.
`asParts`	Returns an Array of six SmallIntegers (year month day hours minutes seconds) that expresses the receiver in local time.
`asPartsGmt`	Returns an Array of six SmallIntegers (year month day hours minutes seconds) that expresses the receiver in Greenwich Mean Time.
`asSeconds`	Returns an Integer that represents the receiver in units of seconds since midnight January 1, 1901, Greenwich Mean Time.
`timeAsSeconds`	Returns a SmallInteger (between zero and 86399 inclusive) that gives the number of seconds represented by the receiver since midnight, Greenwich Mean Time.

Formatting

`asString`	Returns a String that expresses the receiver in local time in the default format (DD/MM/YYYY HH:MM:SS).
`asStringGmt`	Returns a String that expresses the receiver in Greenwich Mean Time in the default format (DD/MM/YYYY HH:MM:SS).

`asStringGmtUsingFormat:` *anArray*

Returns a String that expresses the receiver in Greenwich Mean Time in the format defined by *anArray*. Generates an error if *anArray* contains an incorrect formatting specification.

See Table 2.2 for a complete description of the String-formatting specification Array.

`asStringUsingFormat:` *anArray*

Returns a String that expresses the receiver in local time in the format defined by *anArray*. Generates an error if *anArray* contains an incorrect formatting specification.

See Table 2.2 for a complete description of the String-formatting specification Array.

`US12HrFormat` Returns a String that expresses the receiver in local time. The date is in United States format (month first) and the time of day is based on the 12-hour clock (MM/DD/YY HH:MM:SS pm).

`US24HrFormat` Returns a String that expresses the receiver in local time. The date is in United States format (month first) and the time of day is based on the 24-hour clock (MM/DD/YY HH:MM:SS).

Storing and Loading

`writeTo:` *passiveObj* Writes the passive form of the receiver into *passiveObj*, expressed in Greenwich Mean Time.

Class Protocol

Instance Creation

`fromStream:` *aStream*
Creates and returns an instance of the receiver by reading a String from *aStream*. The String expresses local time in the default format (DD/MM/YYYY HH:MM:SS). Generates an error if the String does not conform to the format.

`fromStream:` *aStream* `usingFormat:` *anArray*
Creates and returns an instance of the receiver by reading a String from *aStream*. The String expresses local time in the format specified by *anArray*. The expression is terminated either by a space character or by the end of the Stream. Generates an error if the String does not conform to the format, or if *anArray* contains an incorrect formatting specification.

See Table 2.2 for a complete description of the String-formatting specification Array.

If the month format (5th element) indicates either an abbreviation (2) or an entire name (3), then this method tries to determine the month by decoding a character substring. That substring may include any number of characters, but must exactly match a legal month name (or the initial portion of that month name). If the substring matches more than one month, the first month matched is used (the search begins with January).

If the specification indicates that seconds should not be included (9th element is false), and aString includes seconds, this method generates an error.

`fromStreamGmt:` *aStream*
Creates and returns an instance of the receiver by reading a String from *aStream*. The String expresses Greenwich Mean Time in the default format (DD/MM/YYYY HH:MM:SS). Generates an error if the String does not conform to the format.

`fromStreamGmt:` *aStream* `usingFormat:` *anArray*

Creates and returns an instance of the receiver by reading a String from *aStream*. The String expresses Greenwich Mean Time in the format specified by *anArray*. The expression is terminated either by a space character or by the end of the Stream. Generates an error if the String does not conform to the format, or if *anArray* contains an incorrect formatting specification.

See Table 2.2 for a complete description of the String-formatting specification Array.

If the month format (5th element) indicates either an abbreviation (2) or an entire name (3), then this method tries to determine the month by decoding a character substring. That substring may include any number of characters, but must exactly match a legal month name (or the initial portion of that month name). If the substring matches more than one month, the first month matched is used (the search begins with January).

If the specification indicates that seconds should not be included (9th element is false), and aString includes seconds, this method generates an error.

`fromString:` *aString*

Creates and returns an instance of the receiver from the String *aString*. The String expresses local time in the default format (DD/MM/YYYY HH:MM:SS). Generates an error if the String does not conform to the format.

`fromString:` *aString* `usingFormat:` *anArray*

Creates and returns an instance of the receiver from the String *aString*. The String expresses local time in the format specified by *anArray*. The expression is terminated either by a space character or by the end of the String. Generates an error if the String does not conform to the format, or if *anArray* contains an incorrect formatting specification.

See Table 2.2 for a complete description of the String-formatting specification Array.

If the month format (5th element) indicates either an abbreviation (2) or an entire name (3), then this method tries to determine the month by decoding a character substring. That substring may include any number of characters, but must exactly match a legal month name (or the initial portion of that month name). If the substring matches more than one month, the first month matched is used (the search begins with January).

If the specification indicates that seconds should not be included (9th element is false), and *aString* includes seconds, this method generates an error.

`fromStringGmt:` *aString* Creates and returns an instance of the receiver from the String *aString*. The String expresses Greenwich Mean Time in the default format (DD/MM/YYYY HH:MM:SS). Generates an error if the String does not conform to the format.

`fromStringGmt:` *aString* `usingFormat:` *anArray*

Creates and returns an instance of the receiver from the String *aString*. The String expresses Greenwich Mean Time in the format specified by *anArray*. The expression is terminated either by a space character or by the end of the String. Generates an error if the String does not conform to the format, or if *anArray* contains an incorrect formatting specification.

See Table 2.2 for a complete description of the String-formatting specification Array.

If the month format (5th element) indicates either an abbreviation (2) or an entire name (3), then this method tries to determine the month by decoding a character substring. That substring may include any number of characters, but must exactly match a legal month name (or the initial portion of that month name). If the substring matches more than one month, the first month matched is used (the search begins with January).

If the specification indicates that seconds should not be included (9th element is false), and *aString* includes seconds, this method generates an error.

`newGmtWithYear:` *year* `dayOfYear:` *dayCount* `seconds:` *seconds*

Creates and returns an instance of the receiver from the specified values, which express Greenwich Mean Time.

Generates an error if any of the values are out of range. The argument *year* must be a positive Integer between 1901 and 1,000,000 inclusive.

`newGmtWithYear:` *yearInt* `month:` *monthInt* `day:` *dayInt* `hours:` *hourInt* `minutes:` *minuteInt* `seconds:` *secondInt*

Creates and returns an instance of the receiver from the specified values, which express Greenwich Mean Time.

Generates an error if any of the values are out of range. The argument *yearInt* must be a positive Integer between 1901 and 1,000,000 inclusive.

`newWithYear:` *year* `dayOfYear:` *dayCount* `seconds:` *seconds*

Creates and returns an instance of the receiver from the specified values, which express local time.

Generates an error if any of the values are out of range. The argument *year* must be a positive Integer between 1901 and 1,000,000 inclusive.

`newWithYear:` *yearInt* `month:` *monthInt* `day:` *dayInt* `hours:` *hourInt* `minutes:` *minuteInt* `seconds:` *secondInt*

Creates and returns an instance of the receiver from the specified values, which express local time.

Generates an error if any of the values are out of range. The argument *yearInt* must be a positive Integer between 1901 and 1,000,000 inclusive.

`now`

Creates and returns an instance of the receiver from the system calendar and clock on the machine that is running the Gem process, which are assumed to represent the current date and time of day expressed in Greenwich Mean Time.

Storing and Loading

`loadFrom:` *passiveObj*

Creates and returns an active instance of the receiver from the passive form of the object, which expresses itself in Greenwich Mean Time.

DecimalFloat

This class represents base 10 floating point numbers, as defined in IEEE standard 854-1987.

You may not create subclasses of DecimalFloat.

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

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

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

Superclasses	Number, Magnitude, Object
Named Instance Variables	None
Instance Format	Byte, Indexable, Variant
Subclass Creation	Disallowed

Instance Protocol

Accessing

`at:` *anIndex* `put:` *aValue*
: Disallowed. You may not change the value of a DecimalFloat.

`denominator`
: Returns the denominator of a Fraction representing the receiver.

`numerator`
: Returns the numerator of a Fraction representing the receiver.

`sign`
: Returns 1 if the receiver is greater than zero, -1 if the receiver is less than zero, and zero if the receiver is zero.

`size:` *anInteger*
: Disallowed. You may not change the size of a DecimalFloat.

Arithmetic

`*` *aNumber*
: Returns the result of multiplying the receiver by *aNumber*.

`+` *aNumber*
: Returns the sum of the receiver and *aNumber*.

`–` *aNumber*
: Returns the difference between the receiver and *aNumber*.

`/` *aNumber*
: Returns the result of dividing the receiver by *aNumber*.

`//` *aNumber*
: Divides the receiver by *aNumber*. Returns the integer quotient, with truncation toward negative infinity. For example,

```
9//4 = 2
-9//4 = -3
-0.9//0.4 = -3
```

The selector `\\` returns the remainder from this division.

`abs`
: Returns a Number that is the absolute value of the receiver.

`factorial`
: Returns the factorial of the integer part of the receiver. Returns 1 if the receiver is less than or equal to 1.

`negated`
: Returns a Number that is the negation of the receiver.

`rem:` *aNumber* — Returns the integer remainder defined in terms of quo: (division of the receiver by *aNumber*, with truncation toward zero).

`sqrt` — Returns the square root of the receiver.

Comparing

`<` *aNumber* — Returns true if the receiver is less than *aNumber*, and false otherwise.

`<=` *aNumber* — Returns true if the receiver is less than or equal to a *aNumber*, and false otherwise.

`=` *aNumber* — Returns true if the receiver is equal to *aNumber*, and false otherwise.

`>=` *aNumber* — Returns true if the receiver is greater than or equal to *aNumber*, and false otherwise.

`~=` *aNumber* — Returns true if the receiver is not equal to *aNumber*, and false otherwise.

Converting

`asDecimalFloat` — Returns the receiver.

`asFloat` — Returns a Float whose value is represented by the receiver.

`asFraction` — Returns a Fraction that represents the receiver. If the receiver is a NaN, or Infinity, returns the receiver.

Formatting

`asString` — Returns a String corresponding to the receiver. Where applicable, returns one of the following Strings: PlusInfinity, MinusInfinity, PlusQuietNaN, MinusQuietNaN, PlusSignalingNaN, or MinusSignalingNaN.

Note:
GemStone currently formats DecimalFloats independently. Specifically, it does not adjust the style of representation according to locale (decimal notation is not internationalized). Under some circumstances, string representation of DecimalFloats may be inconsistent with those of Floats (which do internationalize).

`asStringUsingFormat:` *anArray*

Returns a String corresponding to the receiver, using the format specified by *anArray*. The Array contains three elements: two Integers and a Boolean. Generates an error if any element of the Array is missing or is of the wrong class.

The first element of the Array (an Integer) specifies a minimum number of characters in the result String (that is, the width of the string). If this element is positive, the resulting String is padded with blanks to the right of the receiver. If this element is negative, the blanks are added to the left of the receiver. If the value of this element is not large enough to completely represent the DecimalFloat, a longer String will be generated.

The second element of the Array (also an Integer) specifies the maximum number of digits to display to the right of the decimal point. If the value of this element exceeds the number of digits required to completely specify the DecimalFloat, only the required number of digits are actually displayed. If the value of this element is insufficient to completely specify the DecimalFloat, the value of the DecimalFloat is rounded up or down before it is displayed.

The third element of the Array (a Boolean) indicates whether or not to display the magnitude using exponential notation. (The value true indicates exponential notation and false indicates decimal notation.)

For example, the number 12.3456 displayed with two different format arrays would appear as follows:

Format	Output
`#(10 5 true)`	`' 1.23456E1'`
`#(10 2 false)`	`'12.34'`

Note:
GemStone currently formats DecimalFloats independently. Specifically, it does not adjust the style of representation according to locale (decimal notation is not internationalized). Under some circumstances, string representation of DecimalFloats may be inconsistent with those of Floats (which do internationalize).

Storing and Loading

`writeTo:` *passiveObj*	Converts the receiver to its passive form and writes that information on *passiveObj*.

Testing

`even`	Returns true if the receiver is an even integer, false otherwise.
`odd`	Returns true if the receiver is an odd integer, false otherwise.

Truncation and Rounding

`ceiling`	Returns the integer that is closest to the receiver, on the same side of the receiver as positive infinity.
`floor`	Returns the integer that is closest to the receiver, on the same side of the receiver as negative infinity.
`fractionPart`	Returns the decimal part of the receiver.
`integerPart`	Returns an integer representing the receiver truncated toward zero.
`rounded`	Returns the integer nearest in value to the receiver.
`roundTo:` *aNumber*	Returns the multiple of *aNumber* that is nearest in value to the receiver.
`truncated`	Returns the integer that is closest to the receiver, on the same side of the receiver as zero is located. In particular, returns the receiver if the receiver is an integer.
`truncateTo:` *aNumber*	Returns the multiple of *aNumber* that is closest to the receiver, on the same side of the receiver as zero is located. In particular, returns the receiver if the receiver is a multiple of *aNumber*.

Class Protocol

Arithmetic

`pi` Returns the value of pi, accurate to twenty decimal places.

Converting

`fromString:` *aString* Returns an instance of DecimalFloat, constructed from *aString*. The String must contain only characters representing the object to be created, although leading and trailing blanks are permitted.

The exponent notation, if present may start with any one of $e, $E, $f, or $F.

Exception Handling

Float status flags, exception handlers, and non-default rounding modes are maintained only for a single GemStone Smalltalk execution, and are cleared when a new execution begins.

`clearAllExceptions` Clears all raised exceptions.

`clearException:` *aString*
Clears the raised exception type defined by aSymbol (#divideByZero, #inexactResult, #invalidOperation, #overflow, #underflow). If aSymbol is not one of these exception types, an error is generated. Raised exceptions are set by GemStone during floating point operations, and must be explicitly cleared with this method.

`enabledExceptions` Returns a list of all raised exceptions.

`on:` *aString* `do:` *aBlock* Has no effect in GemStone V5.0.

`operationException:` *aString*
Returns true if the specified exception has occurred in the current operation. Otherwise, returns false. The argument *aString* defines the exception type (#divideByZero, #inexactResult, #invalidOperation, #overflow, #underflow). If aSymbol is not one of these, an error is generated.

`operationExceptions` Returns a list of all exceptions raised by the last floating point operation.

`raisedException:` *aString*

Returns true if the specified exception has occurred since the last `clearException:` operation. Otherwise, returns false. The argument aSymbol defines the exception type (#divideByZero, #inexactResult, #invalidOperation, #overflow, #underflow). If aSymbol is not one of these, an error is generated.

The occurrence of a floating point exception that is not trapped by `on:do:` causes that exception to be raised.

`raisedExceptions` Returns a list of all raised exceptions.

`status` Returns a six-element array. The first element of the Array is a String representing the status of the floating point processor, including the operation exceptions, raised exceptions, rounding mode, and the enabled traps. The next five elements of the Array are the blocks associated with each of the enabled traps, in this order: divideByZero, inexactResult, invalidOperation, overflow, underflow.

Any method that modifies the trap handlers should first save the status using this method. After the method has modified the trap handlers, it should use status: to restore the status.

`status:` *aString* Restores the status of the floating point processor to the previously saved status represented by aSymbol. The argument aSymbol is the first element of the Array that `DecimalFloat | status` returns.

`trapEnabled:` *aString* Returns true if a trap handler has been defined for the specified exception Otherwise, returns false.

Instance Creation

`fromStream:` *aStream*

Generates a DecimalFloat from *aStream*. Generates an error if an attempt is made to read beyond the end of the stream.

The Stream must contain a legal DecimalFloat, as defined by the following BNF construction:

```
DecimalFloat =
    ( Integer '.' Digit {Digit}
      [ E Integer ] ) |
    ( Integer E Integer )
Integer = [ ('+' | '-') ] Digit {Digit}
E = ( 'E' | 'e')
```

Note that the syntax does not allow certain valid DecimalFloats (such as DecimalPlusInfinity and MinusInfinity) to be read.

`new`

Returns a PlusSignalingNaN. You can use this method to define a DecimalFloat without specifying its value.

Storing and Loading

`loadFrom:` *passiveObj*

Reads from *passiveObj* the passive form of an object. Converts the object to its active form by loading the information into a new instance of the receiver. Returns the new instance.

Truncation and Rounding

`roundingMode`

Returns the current rounding mode (nearestEven, towardMinusInfinity, towardPlusInfinity, towardZero).

`roundingMode:` *aString*

The argument *aString* defines the rounding mode (nearestEven, towardMinusInfinity, towardPlusInfinity, towardZero). If *aString* is not one of these, an error is generated.

Dictionary

Dictionary is a concrete subclass of AbstractDictionary. In each Dictionary, all keys should be of the same class.

A Dictionary stores key-value pairs as instances of class Association, and is therefore a collection of Associations. As a result, a Dictionary has two kinds of instance protocols:

- Methods that view the Dictionary as key/value pairs.
- Methods that involve the Association objects themselves.

A Dictionary is also an equality-based collection. That is, two keys or two values are considered to be the same if they are equivalent; they need not be identical to be the same. Thus, if you add two key-value pairs to a Dictionary but the keys are equivalent, even if they are not identical, then the result is that the second pair overwrites the first one, because the keys are the same.

Some other kinds of dictionaries do not store key-value pairs as Associations. Still other kinds are identity-based rather than equality-based. These other kinds of dictionaries exhibit better performance than Dictionary and are to be preferred where they are appropriate.

Warning:
Do not implement subclasses of Dictionary that use the implemenation of Dictionary and compare keys by Identity. All identity based dictionary classes must be a subclass of IdentityKeyValueDictionary or IdentityDictionary in order for GemStone's Symbol canonicalization to work properly.

Superclasses

AbstractDictionary, Collection, Object

Named Instance Variables

count — A SmallInteger, the number of Associations in the instance.

tableSize — A SmallInteger, the size of an internal table for storing elements.

emptySlotHint — A SmallInteger, for GemStone internal use.

numEmptySlots — A SmallInteger, for GemStone internal use.

constraint — A Class, the constraint for the Associations in the instance.

Instance Format Pointer, Indexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`associationAt:` *aKey* Returns the Association with key *aKey*. Generates an error if no such Association exists.

`associationAt:` *aKey* `ifAbsent:` *aBlock*
Returns the Association with key *aKey*. If no such Association exists, returns the result of evaluating the zero-argument block *aBlock*.

`associationAt:` *aKey* `otherwise:` *defaultValue*
Returns the Association with key *aKey*. If no such Association exists, returns the given default value.

`at:` *aKey* Returns the value of the Association with key *aKey*. Reports an error if no such key exists.

`at:` *aKey* `ifAbsent:` *aBlock*
Returns the value of the Association with key *aKey*. If no such Association exists, returns the result of evaluating the zero-argument block *aBlock*.

`at:` *aKey* `otherwise:` *defaultValue*
Returns the value of the Association with key *aKey*. If no such Association exists, returns *defaultValue*.

`constraint` Returns value of the instance variable **constraint**.

`keyAtValue:` *anObject* `ifAbsent:` *aBlock*
Returns the key of the first value equal to the given object, *anObject*. If no match is found, evaluates and returns the result of the block *aBlock*.

`size` Returns the number of Associations in the receiver.

Adding

`add:` *anAssociation*
Adds the association or to the receiver. If the receiver already includes an association/key-value pair whose key is equal to that of *anAssociation*, then this method redefines the value portion of that Association/key-value pair. Returns *anAssociation*.

Removing

`removeAssociation:` *anAssociation*
Removes an element from the receiver equal to *anAssociation* and returns *anAssociation*. If no such element is present, this method generates an error.

`removeAssociation:` *anAssociation* `otherwise:` *defaultValue*
Removes an element from the receiver equal to *anAssociation* and returns *anAssociation*. If an element equal to *anAssociation* is not present, returns *defaultValue*.

`removeKey:` *aKey* `otherwise:` *defaultValue*
Removes an element from the receiver with key *aKey* and returns an Association. If an element with key *aKey* is not present, returns *defaultValue*.

Storing and Loading

`basicWriteTo:` *passiveObj*
Converts the receiver to its passive form and writes that information on *passiveObj*.

`loadVaryingFrom:` *passiveObj* `size:` *varyingSize*
Reads the varying part of the receiver from the given passive object. Does not record the receiver as having been read. Does not read the receiver's named instvars, if any.

Updating

`constraint:` *aClass* Update the value of the instance variable **constraint**.

Class Protocol

Accessing the Class Format

`firstPublicInstVar` Returns the index of the first user-visible instance variable defined in this class, regardless of whether or not this class actually has user-visible instance variables.

`hasPublicInstVars` Returns true if this class has user-visible instance variables defined, false if not.

Instance Creation

`new` Returns a new instance of Dictionary.

`new:` *count* Returns a new instance of Dictionary. The argument *count* provides a hint of the number of elements the instance should be designed to hold.

count should be a SmallInteger.

DoubleByteString

A DoubleByteString is a string for which each character occupies two bytes. The first byte of each character in a DoubleByteString is the more significant byte; the second character is the less significant byte.

DoubleByteString is in the classHistory of String, so instances of DoubleByteString may be stored into instance variables that are constrained to hold instances of String. The inverse is not true, so in an application that uses a mixture of DoubleByteStrings and Strings, string constraints should always be expressed as String.

Superclasses CharacterCollection, SequenceableCollection, Collection, Object

Named Instance Variables None

Instance Format Byte, Indexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`at:` *anIndex* Returns the Character at *anIndex*.

`numArgs` Returns the number of arguments the receiver would take, if the receiver were a message selector.

`size` Returns the number of double-byte characters in the receiver, which is also half the number of bytes that the receiver occupies.

`valueAt:` *anIndex* Returns the value of the (double-byte) character at *anIndex*.

`wordAt:` *anIndex* Returns the integer value of the Character at *anIndex* in the receiver.

Adding

`addAll:` *aCharOrCharColl*

Equivalent to `add:` *aCharOrCharColl*.

`addLast:` *aCharOrCharColl*

Equivalent to `add:` *aCharOrCharColl*.

`insertAll:` *aCharOrCharColl* `at:` *anIndex*

Inserts *aCharOrCharColl* at the specified index. Returns *aCharOrCharColl*.

Case-Insensitive Comparisons

`<` *aCharCollection*

Returns true if the receiver collates before the argument. Returns false otherwise.

The comparison is case-insensitive.

`>` *aCharCollection*

Returns true if the receiver collates after the argument. Returns false otherwise.

The comparison is case-insensitive.

`at:` *anIndex* `equalsNoCase:` *aCharCollection*

Returns true if *aCharCollection* is contained in the receiver, starting at *anIndex*. Returns false otherwise. The comparison is case-insensitive.

`equalsNoCase:` *aCharCollection*

Returns true if corresponding characters in the receiver and argument are equal and *aCharCollection* is comparable with the receiver, and *aCharCollection* is not a kind of Symbol. Returns false otherwise.

The comparison for equal is case-insensitive.

Note that 'kind of Symbol' means either an instance of Symbol or instance of DoubleByteSymbol.

`isEquivalent:` *aCharCollection*

Returns true if corresponding characters in the receiver and argument are equal and *aCharCollection* is comparable with the receiver, and aString is not a kind of Symbol. Returns false otherwise.

The comparison for equal is case-insensitive.

Note that 'kind of Symbol' means either an instance of Symbol or instance of DoubleByteSymbol.

Case-Sensitive Comparisons

= *aCharCollection*

Returns true if corresponding characters in the receiver and argument are equal and *aCharCollection* is comparable with the receiver, and *aCharCollection* is not a kind of Symbol. Returns false otherwise.

The comparison for equal is case-sensitive .

Note that 'kind of Symbol' means either an instance of Symbol or instance of DoubleByteSymbol.

`at:` *anIndex* `equals:` *aCharCollection*

Returns true if *aCharCollection* is contained in the receiver, starting at *anIndex*. Returns false otherwise. The comparison is case-sensitive.

Case-Sensitive Searching

`includesValue:` *aCharacter*

Returns true if the receiver contains *aCharacter*. The search is case-sensitive.

`indexOf:` *aCharacter* `startingAt:` *startIndex*

Returns the index of the first occurrence of *aCharacter* in the receiver, not preceding *startIndex*. If the receiver does not contain *aCharacter*, returns zero.

The search is case sensitive.

Concatenating

, *aCharOrCharColl*	Returns a new instance of the receiver's class that contains the elements of the receiver followed by the elements of *aCharOrCharColl*. The argument must be a String, a DoubleByteString, or an AbstractCharacter. The result may not be an instance of the class of the receiver if one of the following rules applies: 1. If the receiver or argument is a kind of Symbol, DoubleByteSymbol, or ObsoleteInvariantString, the result is a DoubleByteString. 2. If the argument is not a DoubleByteString, and is a subclass of DoubleByteString, then the result is an instance of the class of the argument. Warning: Creating a new instance and copying the receiver take time. If you can safely modify the receiver, it can be much faster to use the `addAll:` method. See the documentation of the Concatenating category of class SequenceableCollection for more details.

Converting

`asDoubleByteSymbol`	Returns a symbol representation of the receiver.
`asString`	Returns a (single byte) String representation of the receiver if all of the characters in the receiver can be represented as single byte characters. Otherwise, returns the receiver.
`asSymbol`	Returns a canonical Symbol representation of the receiver.
`asSymbolKind`	Returns a (single byte) Symbol representation of the receiver.
`asUppercase`	Returns a new instance of the receiver's class, with all lowercase characters in the receiver changed to uppercase. Lower case characters are the ASCII characters $a to $z inclusive. If the receiver is a DoubleByteSymbol, returns an instance of DoubleByteString.

Copying

`copyFrom:` *index1* `to:` *index2* `into:` *anObject* `startingAt:` *index3*

Copies the elements of the receiver between *index1* and *index2*, inclusive, into *anObject* starting at *index3*, overwriting the previous contents. If *anObject* is the same object as the receiver, the source and destination blocks may overlap.

The argument *anObject* should be a SequenceableCollection.

`withLFs` Returns a copy of the receiver with all back-slashes replaced by line-feeds.

Execution

`evaluate` Compiles the receiver as an unbound method and executes it using the current default symbol list.

`evaluateInContext:` *anObject* `symbolList:` *aSymbolList*

Compiles the receiver as an instance method for the class of *anObject*, using *aSymbolList* as the symbol list. Executes the resulting GsMethod using *anObject* as self and returns the result of the execution. Generates an error if compilation errors occur.

Hashing

`hash` Returns a positive Integer based on a case-sensitive hash of the contents of the receiver.

Other Comparisons

`asciiLessThan:` *aString* Returns true if the receiver collates before the argument using the ASCII collating table, which collates AB...Z...ab..z.

`equals:` *aString* `collatingTable:` *aTable*

Returns true if the receiver collates the same as the argument.

The *aString* argument may be a String or a DoubleByteString.

The collating sequence is defined by *aTable*, which must be an instance of DoubleByteString. The argument *aTable* is accessed with `wordAt:` to retrieve the collating sequence for a particular character. Use `wordAt:put:` to populate it. Characters beyond the range of *aTable* are collated using their raw character value.

`greaterThan:` *aString* `collatingTable:` *aTable*

Returns true if the receiver collates after the argument.

The *aString* argument may be a String or a DoubleByteString.

The collating sequence is defined by *aTable*, which must be an instance of DoubleByteString. The argument *aTable* is accessed with `wordAt:` to retrieve the collating sequence for a particular character. Use `wordAt:put:` to populate it. Characters beyond the range of *aTable* are collated using their raw character value.

`greaterThanOrEqual:` *aString* `collatingTable:` *aTable*

Returns true if the receiver collates after or the same as the argument.

The *aString* argument may be a String or a DoubleByteString.

The collating sequence is defined by *aTable*, which must be an instance of DoubleByteString. The argument *aTable* is accessed with `wordAt:` to retrieve the collating sequence for a particular character. Use `wordAt:put:` to populate it. Characters beyond the range of *aTable* are collated using their raw character value.

`lessThan:` *aString* `collatingTable:` *aTable*

Returns true if the receiver collates before the argument.

The *aString* argument may be a String or a DoubleByteString.

The collating sequence is defined by *aTable*, which must be an instance of DoubleByteString. The argument *aTable* is accessed with `wordAt:` to retrieve the collating sequence for a particular character. Use `wordAt:put:` to populate it. Characters beyond the range of *aTable* are collated using their raw character value.

`lessThanOrEqual:` *aString* `collatingTable:` *aTable*

Returns true if the receiver collates before or the same as the argument.

The *aString* argument may be a String or a DoubleByteString.

The collating sequence is defined by *aTable*, which must be an instance of DoubleByteString. The argument *aTable* is accessed with `wordAt:` to retrieve the collating sequence for a particular character. Use `wordAt:put:` to populate it. Characters beyond the range of *aTable* are collated using their raw character value.

Storing and Loading

`writeTo:` *passiveObj* Converts the receiver to its passive form and writes that information on *passiveObj*.

Updating

`size:` *anInteger* Changes the size of the receiver to *anInteger*.

If *anInteger* is less than the current size of the receiver, the receiver is shrunk accordingly. If *anInteger* is greater than the current size of the receiver, the receiver is extended and new elements are initialized to nil.

`wordAt:` *anIndex* `put:` *aValue*

Stores the integer value *aValue* in the character cell of the receiver specified by *anIndex*. Return *aValue*.

Class Protocol

Instance Creation

`withAll:` *aString*	Returns an instance of the receiver containing the elements of the argument.

Storing and Loading

`loadFrom:` *passiveObj*	Reads from *passiveObj* the passive form of an object. Converts the object to its active form by loading the information into a new instance of the receiver. Returns the new instance.

DoubleByteSymbol

A DoubleByteSymbol is an invariant String for which all comparisons are case-sensitive. DoubleByteSymbols are used internally to represent variable names and selectors. DoubleByteSymbols are always invariant and cannot be modified at any time after they are created. Hence, the new and new: methods are disallowed.

All Symbols and DoubleByteSymbols are canonical, which means that it is not possible to create two of them that have the same value. If two canonical symbols compare as equal, then they are the same (identical) object. Every instance of DoubleByteSymbol will contain at least one Character whose value is greater than 255. A Symbol whose character values are all less than 256 is always an instance of Symbol.

GemStone places all canonical symbols in the DataCuratorSegment. However, GemStone does permit you to commit a canonical Symbol, even if you have no explicit write authorization for the DataCuratorSegment. GemStone also gathers all canonical symbols into one collection (a CanonicalStringDictionary) called AllSymbols, which it also places in the DataCuratorSegment.

Since canonical symbols are universally visible, it is not recommended that they be used for names that should remain private or secure. Such objects should be instances of InvariantString instead.

Since canonical symbols must be universally available, you cannot lock a Symbol or DoubleByteSymbol.

Since each canonical symbol has a unique value, you cannot copy a Symbol or DoubleByteSymbol. In addition, to guarantee canonicalization, you cannot send the become: or changeClassTo: messages to a Symbol or DoubleByteSymbol.

DoubleByteSymbol is in the classHistory of Symbol, so instances of DoubleByteSymbol may be stored into instance variables that are constrained to hold instances of Symbol. The inverse is not true, so you should always express symbol constraints as Symbol.

EUCSymbols are not canonicalized and cannot be used interchangeably with canonical symbols. They do not satisfy a constraint of Symbol, and are not accepted by the virtual machine as message selectors.

Superclasses	DoubleByteString, CharacterCollection, SequenceableCollection, Collection, Object
Named Instance Variables	None
Instance Format	Byte, Indexable, Variant
Subclass Creation	Disallowed

Instance Protocol

Comparing

= *anObject*	Returns true if *anObject* is equal to the receiver. Since symbols and double byte symbols are canonicalized, this method does the check based on the identities of the receiver and the argument.
`hash`	Returns a numeric hash key for the receiver.

Concatenating

, *aCharOrCharCollection*	Returns a new instance of DoubleByteString that contains the elements of the receiver followed by the elements of *aCharOrCharCollection*. A DoubleByteString is returned rather than a DoubleByteSymbol to avoid the expense of unnecessary creation and canonicalization of Symbols.

Converting

`asDoubleByteString`	Returns a copy of the receiver as a DoubleByteString.
`asDoubleByteSymbol`	Returns the receiver.
`asString`	Returns a copy of the receiver as a String.
`asSymbol`	Returns the reciever. All Symbols and DoubleByteSymbols are canonical.
`asSymbolKind`	Returns the reciever. All Symbols and DoubleByteSymbols are canonical.

Copying

`copy`	Returns self. Copies of (canonical) double byte symbols are not allowed.

Class Protocol

Instance Creation

`new`	Disallowed. To create a new DoubleByteSymbol, use the class method `withAll:` instead.
`new:` *size*	Disallowed. To create a new DoubleByteSymbol, use the class method `withAll:` instead.
`withAll:` *aString*	Returns a canonical symbol that has the same Characters as *aString*. Returns an existing canonical symbol if it is already in AllSymbols, or if it was created earlier in the current session. Otherwise, creates and returns a new canonical symbol. The canonical symbol that this method returns is a Symbol if *aString* is a DoubleByteString with all character values less than 255. Otherwise, it returns a DoubleByteSymbol.

Storing and Loading

`loadFrom:` *passiveObj*	Reads from *passiveObj* the passive form of an object. Converts the object to its active form by loading the information into a new instance of the receiver. Returns the new instance.

EUCString

This class represents Japanese strings in Extended Unix Code format.

Superclasses JapaneseString, CharacterCollection, SequenceableCollection, Collection, Object

Named Instance Variables None

Instance Format Byte, Indexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`at:` *anIndex* Returns the JISCharacter at the specified index.

`size` Returns the number of JISCharacters in the receiver.

Adding

`add:` *aCharOrCharCollection*

Appends *aCharOrCharCollection* to the receiver. The argument *aCharOrCharCollection* must be a CharacterCollection or a kind of AbstractCharacter.

`addAll:` *aCharOrCharCollection*

Equivalent to `add:` *aCharOrCharCollection*.

`addLast:` *aCharOrCharCollection*

Equivalent to `add:` *aCharOrCharCollection*.

`insertAll:` *aCharOrCharCollection* `at:` *anIndex*

Inserts *aCharOrCharCollection* at the specified index.

Comparing

< *aCharCollection*	Returns true if the receiver collates before the argument. Returns false otherwise.
<= *aCharCollection*	Returns true if the receiver collates before the argument or if all of the corresponding characters in the receiver and argument are equal. Returns false otherwise.
= *aCharCollection*	Returns true if all of the corresponding characters in the receiver and argument are equal. Returns false otherwise.
> *aCharCollection*	Returns true if the receiver collates after the argument. Returns false otherwise.
>= *aCharCollection*	Returns true if the receiver collates after the argument or if all of the corresponding characters in the receiver and arguments are equal. Returns false otherwise.

Converting

`asArrayOfPathTerms`	Returns an array of path substrings held by the receiver. The receiver is assumed to be a period-separated list of substrings. These substrings are extracted and collected in an Array. If the receiver contains no periods, the array will hold a copy of the receiver. Periods not meant to separate path terms may be escaped with a backslash character.
`asSymbol`	Returns a Symbol that contains the same bytes as the receiver.
`asSymbolKind`	Returns a copy of the receiver as an instance of Symbol.

Copying

`copyFrom:` *index1* `to:` *index2* `into:` *aSeqColl* `startingAt:` *destIndex*

Copies the elements of the receiver between *index1* and *index2*, inclusive, into *aSeqColl* starting at *destIndex*, overwriting the previous contents. If *aSeqColl* is the same object as the receiver, the source and destination blocks may overlap.

Formatting

`asEUCString` Returns the receiver.

`printOn:` *aStream* Puts a displayable representation of the receiver on the given stream.

`printString` Returns an EUCString whose contents are a displayable representation of the receiver.

Searching

`indexOf:` *aCharacter* `startingAt:` *startIndex*

Returns the index of the first occurrence of *aCharacter* in the receiver, not preceding *startIndex*. If the receiver does not contain *aCharacter*, this returns zero.

Updating

`at:` *anIndex* `put:` *aCharacter*

Stores *aCharacter* at the specified location.

`size:` *anInteger* Changes the size of the receiver to *anInteger*.

If *anInteger* is less than the current size of the receiver, the receiver is shrunk accordingly. If *anInteger* is greater than the current size of the receiver, the receiver is extended and new elements are initialized to nil.

EUCSymbol

An EUCSymbol represents an invariant Japanese symbol in Extended Unix Code format.

EUCSymbols are not canonicalized like Symbol and DoubleByteSymbol.

An EUCSymbol may not be used as a message selector, and may not be stored into an instance variable constrained to hold Symbols.

Superclasses InvariantEUCString, EUCString, JapaneseString, CharacterCollection, SequenceableCollection, Collection, Object

Named Instance Variables None

Instance Format Byte, Indexable, Invariant

Subclass Creation Allowed

Instance Protocol

Formatting

`printOn:` *aStream* Puts a displayable representation of the receiver on the given stream. That representation conforms to GemStone Smalltalk parsing rules.

Testing

`isSymbol` Returns true.

Exception

An Exception is an object that represents a state to be invoked in the event of an error.

Superclasses

Object

Named Instance Variables

next — The next exception to be invoked, if this exception is part of a chain.

category — A SymbolDictionary (such as GemStoneError) that represents the category that this exception traps. If nil, all errors are trapped.

number — A value that controls which errors are trapped. If it is nil, all errors in the specified category are trapped. If it is a SmallInteger, the error with that number is trapped.

The following errors can never be caught with an Exception and are always given back to the controlling GemBuilder for C (GCI) interface:

- ErrorSymbols at: #rtErrStackLimit
- ErrorSymbols at: #bkupErrRestoreSuccessful
- ErrorSymbols at: #abortErrFinishedObjAudit
- ErrorSymbols at: #rtErrHardBreak
- ErrorSymbols at: #rtErrCommitAbortPending
- ErrorSymbols at: #rtErrUncontinuable
- ErrorSymbols at: #rtErrStep

theBlock — A four-argument block to be executed. The arguments are:

- exception, the exception whose block this is.
- errorDictionary, an error dictionary, instance of SymbolDictionary, such as GemStoneError.
- number, the error number (a SmallInteger).
- args, error arguments (normally an Array).

subtype — A code for further differentiating within a category, used by the floating-point exception mechanism.

Instance Format Pointer, Nonindexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`block` Returns the value of the instance variable **theBlock**.

`category` Returns the value of the instance variable **category**.

`next` Returns the next exception to be invoked (the value of the **next** instance variable).

`next:` *anException* Establishes the next exception to be invoked (sets the value of the **next** instance variable).

`number` Returns the value of the instance variable **number**.

`subtype` Returns the value of the exception's **subtype** instance variable.

Creation

`block:` *aBlock* `category:` *aSymbolDictionary* `number:` *num* `subtype:` *atype*

Initialize the receiver using the arguments. The block passed will receive four arguments when it is invoked:

1. This exception.
2. The category (a SymbolDictionary) of the error.
3. The number of the error.
4. An Array of the arguments to the error.

If *aSymbolDictionary* is nil, all exceptions will be trapped by the block. If *aSymbolDictionary* is not nil, but *num* is nil, all exceptions within the given category will be trapped.

The subtype field is for user convenience. It is used by the Float exception mechanism to distinguish different types of float exceptions.

Invocation

`resignal:` *aSymbolDictionary* `number:` *errNum* `args:` *args*

Resignal this exception down the line, as it were. If execution is continued after a successful resignal, returns the receiver.

Management

`remove`

Search the current GemStone Smalltalk call stack for a method or block context that has the receiver installed, and remove it. The stack is searched by starting with the top method or block context and moving down. Generates an error if the receiver is not installed anywhere in the current GemStone Smalltalk call stack.

Class Protocol

Creation

`block:` *aBlock* `category:` *aCategory* `number:` *num* `subtype:` *atype*

Create an Exception. This creates an Exception but does not install it in the virtual machine. The resulting exception could be chained to an already installed exception by using the result as the argument to next: sent to the already installed exception. Otherwise this method is only useful by other methods within this class.

The block passed will receive four arguments when it is invoked:

1. This exception
2. The category of the error
3. The number of the error
4. An Array of the arguments to the error.

If *aCategory* is nil, all exception will be trapped by the block. If *aCategory* is not nil, but *num* is nil, all exceptions within the given category will be trapped.

The subtype field is for user convenience. It is used by the Float exception mechanism to distinguish different types of floating-point exceptions.

Installing

`category:` *aCategory* `number:` *aNumber* `do:` *aBlock*

This method installs an exception for the top method or block context in the current GemStone Smalltalk call stack. Returns the new exception. The block *aBlock* receives four arguments when it is invoked:

1. This exception
2. The category of the error
3. The number of the error
4. An Array of the arguments to the error.

If *aCategory* is nil, all exceptions are trapped by the block. If *aCategory* is not nil, but *aNumber* is nil, all exceptions within the given category are trapped.

This method must be a primitive, in order for the exception to be installed in the method or block context of the sender.

The subtype of the new exception is nil.

`installDebugException:` *aBlock* `category:` *category* `number:` *num* `subtype:` *atype*

Install the specified exception block as a debug exception block to field errors of the specified category, number, and subtype.

This method is intended for future use in implementing GemStone Smalltalk debuggers, and is not applicable to normal application programming.

`installStaticException:` *aBlock* `category:` *category* `number:` *num*

Install the specified exception block as a static exception block to field errors of the specified category and number.

`installStaticException:` *aBlock* `category:` *category* `number:` *num* `subtype:` *atype*

Install the specified exception block as a static exception block to field errors of the specified category, number, and subtype.

Management

`removeActivationException:` *anException*

Search the current GemStone Smalltalk call stack for a method or block context that has *anException* installed, and remove it. The stack is searched by starting with the top method or block context and moving down. Generates an error if *anException* is not installed anywhere in the current GemStone Smalltalk call stack.

`removeDebugException:` *anException*

Unlink a debug exception.

This method is intended for future use in implementing GemStone Smalltalk debuggers, and is not applicable to normal application programming.

`removeStaticException:` *anException*

Unlink a static exception.

ExecutableBlock

ExecutableBlock is an abstract superclass for the various kinds of GemStone Smalltalk code blocks that can be executed within the object server.

Superclasses

BlockClosure, Object

Named Instance Variables

method — A GsMethod that lexically contains this block.

firstPC — A SmallInteger, zero-based offset from first executable instruction in **method** to first instruction in block.

spare1 — Unused.

numberArgs — A SmallInteger that represents the number of arguments to this block.

numberTemps — A SmallInteger that represents the number of temporaries in this block.

firstSourceOffset — A SmallInteger that gives the 1-based offset into the method's source string of the first character of the block's source (a left square brace).

lastSourceOffset — A SmallInteger that gives the 1-based offset into the method's source string of the last character of the block's source (a right square brace).

argsAndTemps — An InvariantArray that contains Symbols which are the block's argument and temporary names, in the order they are allocated in this block's context. For blocks that have no arguments or temporaries, this is nil.

blockSelfUsed — A Boolean that indicates whether or not ''self'' is used within the block.

Instance Format

Pointer, Nonindexable, Variant

Subclass Creation

Disallowed

Instance Protocol

Accessing

`argsAndTemps` Return the value of the **argsAndTemps** instance variable.

`firstPC` Return the value of the **firstPC** instance variable.

`firstSourceOffset` Return the value of the **firstSourceOffset** instance variable.

`lastSourceOffset` Return the value of the **lastSourceOffset** instance variable.

`method` Return the value of the **method** instance variable.

`numberArgs` Return the value of the **numberArgs** instance variable.

`numberTemps` Return the value of the **numberTemps** instance variable.

Block Evaluation

`valueNowOrOnUnwindDo:` *aBlock*

Evaluate the receiver. Evaluate *aBlock* after evaluating the receiver, or before any return from a block that would return to the sender.

Flow of Control

`untilFalse` (Reserved selector.) Evaluates the receiver repeatedly until the evaluation's result is false. Return nil. Generates an error if the receiver is not a zero-argument block.

`untilTrue` (Reserved selector.) Evaluates the receiver repeatedly until the evaluation's result is true. Return nil. Generates an error if the receiver is not a zero-argument block.

`whileFalse:` *aBlock* (Reserved selector.) Evaluates the zero-argument block *aBlock* repeatedly while the receiver evaluates to false. Return nil. Generates an error if the receiver is not a zero-argument block.

`whileTrue:` *aBlock* (Reserved selector.) Evaluates the zero-argument block *aBlock* repeatedly while the receiver evaluates to true. Return nil. Generates an error if the receiver is not a zero-argument block.

Storing and Loading

`writeTo:` *aPassiveObject* — Converts the receiver to its passive form and writes that information on *aPassiveObject*.

SimpleBlocks can usually be passivated and then reactivated. ComplexBlocks can be passivated but may have to be massaged to be reactivated. References to self in complex blocks will resolve to an instance of Object when the block is activated, and any arguments or temporaries from enclosing scopes will be nil.

Class Protocol

Storing and Loading

`loadFrom:` *passiveObj* — Reads from *passiveObj* the passive form of an object. Converts the object to its active form by loading the information into a new instance of the receiver. Returns the new instance.

Float

This class represents 8 byte binary floating point numbers, as defined in IEEE standard 754.

You may not create subclasses of Float.

The mathematics package of the vendor of the machine where the gem is running implements the numerical behavior of instances of Float.

Each float contains a 64 bit value. The floats are stored on Disk and in object memory in big-endian IEEE format. GemStone Smalltalk primitives and GemBuilder for C (GCI) float conversion functions will automatically convert the format of a float to/from the machines native format as required.

Do not use GciFetch/StoreBytes() directly on a Float. Use GciOopToFlt and GciFltToOop instead. If you choose to violate this rule, and are using an Intel x86 processor, you must convert the format of a float yourself.

The following details are provided for application programmers who choose to access bytes of a float directly.

Starting from the most significant bit of the byte (self _basicAt: 1) , the bits of a Float are as follows:

- 1 bit of sign (0 means positive, 1 means negative)
- 11 bits of exponent
- 52 bits of mantissa

This format, called big-endian, is the native format on Sparc, RS6000, and HP PA-RISC processors.

Here are the 8 byte patterns of the exceptional values:

value	byte 1	byte 2	byte 3	byte 4	byte 5	byte 6	byte 7	byte 8
PlusQuietNaN	16#7f	16#ff	16#ff	16#ff	16#ff	16#ff	16#ff	16#ff
MinusQuietNaN	16#ff	16#ff	16#ff	16#ff	16#ff	16#ff	16#ff	16#ff
MinusInfinity	16#ff	16#f0	0	0	0	0	0	0
MinusSignalingNaN	16#ff	16#f0	0	0	0	0	0	1
PlusSignalingNaN	16#7f	16#f0	0	0	0	0	0	0
PlusSignalingNaN	16#7f	16#f0	0	0	0	0	0	1

If the Float is an exeptional value (a NaN or an Infinity) then the exponent bits are all 1. If in addition the mantissa is 0 then the float is an Infinity; if the mantissa is non-zero then the float is a NaN. In a NaN, if the most significant bit of the Mantissa is 1 then the NaN is a QuietNaN.

Superclasses	BinaryFloat, Number, Magnitude, Object
Named Instance Variables	None
Instance Format	Byte, Indexable, Variant
Subclass Creation	Disallowed

Instance Protocol

Accessing

`denominator`	Returns the denominator of a Fraction representing the receiver.
`numerator`	Returns the numerator of a Fraction representing the receiver.
`sign`	Returns 1 if the receiver is greater than zero, -1 if the receiver is less than zero, and zero if the receiver is zero.

Arithmetic

`*` *aNumber*	Multiply the receiver by *aNumber* and returns the result.
`+` *aNumber*	Returns the sum of the receiver and *aNumber*.
`–` *aNumber*	Returns the difference between the receiver and *aNumber*.
`/` *aNumber*	Divide the receiver by *aNumber* and returns the result.
`arcCos`	Returns the arc-cosine of the receiver in radians.
`arcSin`	Returns the arc-sine of the receiver in radians.
`arcTan`	Returns the arc-tangent of the receiver in radians.
`cos`	Returns the cosine of the receiver which is treated as an angle expressed in radians.
`exp`	Returns e raised to the power of the receiver.
`ln`	Returns the natural logarithm of the receiver.
`log10`	Returns the base 10 logarithm of the receiver.
`raisedTo:` *aNumber*	Returns the receiver raised to the power of the argument.
`sin`	Returns the sine of the receiver which is treated as an angle expressed in radians.
`sqrt`	Returns the square root of the receiver.

`tan` Returns the tangent of the receiver which is treated as an angle expressed in radians.

Comparing

`< aNumber` Returns true if the receiver is less than *aNumber*; returns false otherwise.

`<= aNumber` Returns true if the receiver is less than *aNumber*; returns false otherwise.

`= aNumber` Returns true if the receiver is equal to *aNumber*; returns false otherwise.

`hash` Returns a numerical hash code for the receiver.

`~= aNumber` Returns true if the receiver is not equal to *aNumber*; returns false otherwise.

Converting

`asDecimalFloat` Returns a DecimalFloat representing the receiver.

`asFloat` Returns the receiver.

`asFraction` Returns a Fraction that represents the receiver. If the receiver is a NaN, or Infinity, returns the receiver.

`asSmallFloat` Returns an instance of SmallFloat, which may be a NaN

Formatting

`asString` Returns a String corresponding to the value of the receiver. Where applicable, returns one of the following Strings: PlusInfinity, MinusInfinity, PlusQuietNaN, MinusQuietNaN, PlusSignalingNaN, or MinusSignalingNaN.

`asStringUsingFormat:` *anArray*
Returns a String corresponding to the receiver, using the format specified by *anArray*. The Array contains three elements: two Integers and a Boolean. Generates an error if any element of the Array is missing or is of the wrong class.

The first element of the Array (an Integer between -1000 and 1000) specifies a minimum number of characters in the result String (that is, the width of the string). If this element is positive, the resulting String is padded with blanks to the right of the receiver. If this element is negative, the blanks are added to the left of the receiver. If the value of this element is not large enough to completely represent the Float, a longer String will be generated.

The second element of the Array (a positive Integer less than 1000) specifies the maximum number of digits to display to the right of the decimal point. If the value of this element exceeds the number of digits required to completely specify the Float, only the required number of digits are actually displayed. If the value of this element is insufficient to completely specify the Float, the value of the Float is rounded up or down before it is displayed.

The third element of the Array (a Boolean) indicates whether or not to display the magnitude using exponential notation. (The value true indicates exponential notation and false indicates decimal notation.)

For example, the number 12.3456 displayed with two different format arrays would appear as follows:

Format	Output
`#(10 5 true)`	`' 1.23456E1'`
`#(10 2 false)`	`'12.34'`

Truncation and Rounding

`ceiling`	Returns the integer that is closest to the receiver, on the same side of the receiver as positive infinity.
`floor`	Returns the integer that is closest to the receiver, on the same side of the receiver as negative infinity.
`integerPart`	Returns an integer representing the receiver truncated toward zero.
`rounded`	Returns the integer nearest in value to the receiver.
`roundTo:` *aNumber*	Returns the multiple of *aNumber* that is nearest in value to the receiver.

`truncated`	Returns the integer that is closest to the receiver, on the same side of the receiver as zero is located. If the receiver is an exceptional float (NaN or Infinity) , returns the receiver.
`truncateTo:` *aNumber*	Returns the multiple of *aNumber* that is closest to the receiver, on the same side of the receiver as zero is located. In particular, returns the receiver if the receiver is a multiple of *aNumber*.

Class Protocol

Instance Creation

`fromString:` *aString*	Returns an instance of the receiver, constructed from *aString*. The String must contain only characters representing the object to be created, although leading and trailing blanks are permitted. If the string represents an exceptional float, it must contain one of the following strings, with leading and trailing blanks permitted: PlusInfinity, MinusInfinity, PlusQuietNaN, MinusQuietNaN, PlusSignalingNaN, or MinusSignalingNaN. If the string does not conform to the above rules, an error may be generated, or a SignalingNaN may be returned. If the string is larger than 8191 bytes, an error is generated.
`new`	Returns a PlusSignalingNaN. You can use this method to define a Float without specifying its value.

Storing and Loading

`loadFrom:` *passiveObj*	Reads from *passiveObj* the passive form of an object. Converts the object to its active form by loading the information into a new instance of the receiver. Returns the new instance.

Fraction

A Fraction is a Number represented as the ratio of two Integers.

Superclasses Number, Magnitude, Object

Named Instance Variables **numerator —** The numerator of the Fraction (an Integer).

denominator — The denominator of the Fraction (an Integer).

Instance Format Pointer, Nonindexable, Variant

Subclass Creation Disallowed

Instance Protocol

Accessing

`at:` *anIndex* Disallowed.

`at:` *anIndex* `put:` *aNumber*
Disallowed. You may not change the value of a Fraction.

`denominator` Returns the **denominator** of the receiver.

`instVarAt:` *anIndex* `put:` *aValue*
Disallowed. You may not change the value of a Fraction.

`numerator` Returns the **numerator** of the receiver.

`size:` *anInteger* Disallowed. You may not change the size of a Fraction.

Arithmetic

`*` *aFraction* Returns the result of multiplying the receiver by *aFraction*.

`+` *aFraction* Returns the sum of the receiver and *aFraction*.

`–` *aFraction* Returns the difference between the receiver and *aFraction*.

`/` *aFraction* Returns the result of dividing the receiver by *aFraction*.

`negated` Returns a Number that is the negation of the receiver.

`reciprocal` Returns 1 divided by the receiver. Generates an error if the receiver is 0.

Comparing

`<` *aFraction*	Returns true if the receiver is less than *aFraction*; returns false otherwise.
`<=` *aFraction*	Returns true if the receiver is less than or equal to *aFraction*; returns false otherwise.
`=` *aFraction*	Returns true if the receiver is equal to *aFraction*; returns false otherwise.
`>` *aFraction*	Returns true if the receiver is greater than *aFraction*; returns false otherwise.
`>=` *aFraction*	Returns true if the receiver is greater than *aFraction*; returns false otherwise.
`~=` *aFraction*	Returns true if the receiver is not equal to *aFraction*; returns false otherwise.

Converting

`asDecimalFloat`	Returns an instance of DecimalFloat that has the value of the receiver.
`asFloat`	Returns an instance of Float that has the value of the receiver.
`asScaledDecimal:` *scale*	Returns a ScaledDecimal representation of the receiver.

Formatting

`asString`	Returns a String of the form numerator/denominator.

Storing and Loading

`writeTo:` *passiveObj*	Converts the receiver to its passive form and writes that information on *passiveObj*.

Truncation and Rounding

`truncated`	Returns the integer that is closest to the receiver, on the same side of the receiver as zero is located.

Class Protocol

Instance Creation

Do not send class Fraction the message new. Fractions created in that way are meaningless and cannot be handled properly by GemStone's associative access mechanism. To create a new Fraction, use one of the instance creation methods listed here.

`fromStream:` *aStream*
: Returns a Fraction from the stream. The stream must contain two Integers separated by a slash. (There may be blanks around the slash.) Generates an error if the stream contains anything else, or if an attempt is made to read beyond the end of the stream.

`numerator:` *numInt* `denominator:` *denomInt*
: Returns an instance of Fraction with numerator *numInt* and denominator *denomInt*. If that Fraction can be reduced, this method returns the corresponding Integer instead. The result is made invariant.

  If either argument (numerator or denominator) is not an Integer, that argument is truncated to the corresponding Integer.

Storing and Loading

`loadFrom:` *passiveObj*
: Reads from *passiveObj* the passive form of an object. Converts the object to its active form by loading the information into a new instance of the receiver. Returns the new instance.

GsCurrentSession

GsCurrentSession provides a public interface to the current GemStone session. There is only one instance of GsCurrentSession in each GemStone session. The GemStone server creates it automatically when a user logs into GemStone. The instance is transient and cannot be accessed after the user logs out of GemStone.

Superclasses

GsSession, AbstractSession, Object

Named Instance Variables

symbolList — The SymbolList that is used for default Symbol resolution in the current GemStone session. Its value is initially a copy of the SymbolList of the current user's UserProfile. While its value can be modified during the session, such modifications are transient and last only as long as the current session.

Modifications should be handled with care to avoid unintended side effects in Symbol resolution during the current session. When methods need to create a different transient Symbol resolution environment, it is often preferable to copy this SymbolList first and use the modified copy explicitly where it is needed.

nativeLanguage — A String (normally a Symbol) that controls error message generation in the current session. GemStone automatically executes the message

```
System myUserProfile
   objectNamed: #NativeLanguage
```

at login, and assigns the result to this instance variable. For default UserProfiles, the result is #English.

Instance Format

Pointer, Nonindexable, Variant

Subclass Creation

Disallowed

Instance Protocol

Accessing

`nativeLanguage` Returns the String that designates the language that controls error message generation in the current session.

Accessing the Symbol List

`objectNamed:` *aSymbol*
: Returns the first object in the receiver's symbol list that has the given name. If no object with the given name is found, returns nil.

`resolveSymbol:` *aSymbol*
: Searches the receiver's symbol list for an Association whose key is equal to aString, and returns that Association. If no such Association is found in the symbol list, returns nil.

  Implemented to use the current session's transient copy of the symbol list. This method is the default mechanism for symbol-resolution during compilation of GemStone Smalltalk methods.

`symbolList`
: Implemented to use the current session's transient copy of the symbol list.

Session Configuration Access

`clientVersionAt:` *aSymbol*
: Returns the value of the GsSession's client version information parameter named *aSymbol*. Returns nil if no version parameter named *aSymbol* exists.

`configurationAt:` *aSymbol*
: Returns the value of the configuration parameter named *aSymbol* (for example, #GEM_HALT_ON_ERROR). Raises an error if *aSymbol* is not a valid parameter name.

`configurationAt:` *aSymbol* `put:` *anObject*
: Sets the value of the configuration parameter named *aSymbol* to *anObject*. Raises an error if *aSymbol* is not a valid parameter name or *anObject* is an inappropriate value for the parameter.

`configurationParameters`
: Returns a Set of Symbols containing the names of all valid configuration parameters for this GsSession.

`serverVersionAt:` *aSymbol*
: Returns the value of the GsSession's stone version information parameter named *aSymbol*.

`sessionVersionAt:` *aSymbol*

Returns the value of the GsSession's gem version information parameter named *aSymbol*.

`versionParameters` Returns a Set of Symbols containing the names of all valid version parameters for this GsSession.

Signalling

`signalFromSession` Return a GsInterSessionSignal object containing information about a signal from another session, or nil if there is no signal waiting.

Smalltalk Execution

`execute:` *aString* Executes *aString* containing GemStone Smalltalk code in the session represented by the receiver. Symbol resolution is from the default symbol list.

`execute:` *aString* `symbolList:` *aSymbolList*

Executes *aString* containing GemStone Smalltalk code in the session represented by the receiver. Symbol resolution is from the given symbol list.

Transaction Control

`abortTransaction` Rolls back all modifications made to committed GemStone objects and provides the session with a new view of the most current committed state of the repository.

These operations are performed whether or not the session was previously in a transaction. If the transaction mode is set to #autoBegin, then a new transaction is started. If the transaction mode is set to #manualBegin, then a new transaction is not started.

`beginTransaction` Starts a new transaction for the session. If the session is already in a transaction, aborts the current transaction and starts a new transaction.

If the session changed any permanent objects without committing them, their state is aborted.

`commitAndReleaseLocks`

Attempts to commit the transaction for the session.

This method is the same as `commitTransaction` except for the handling of locks. If the commit succeeds, this method releases all locks for the session and returns true. Otherwise, it returns false and does not release locks.

This method also clears the commit release locks and commit-or-abort release locks sets. See the 'Releasing Locks' method category in class System for more information.

`commitTransaction` Attempts to update the persistent state of GemStone to include changes made by this session.

If the commit operation succeeds, then this method returns true, and the current transaction's changes, if any, become a part of the persistent repository. After the repository update, the session exits the current transaction. If the transaction mode is autoBegin, then the session enters a new transaction. If the transaction mode is manualBegin, then the session remains outside of a transaction.

If conflicts prevent the repository update, then this method returns false. Call the `transactionConflicts` method to determine the nature of the conflicts. If the session is outside of a transaction, then this method raises the error rtErrPrimOutsideTrans.

This method also updates the session's view of GemStone. If the commit operation succeeds, then all objects in the session's view are consistent with the current state of GemStone. If the commit fails, then this method retains all the changes that were made to objects within the current transaction. However, commits made by other sessions are visible to the extent that changes in this transaction do not conflict with them.

`continueTransaction`

Updates the session's view to the most recently committed state of GemStone without rolling back modifications made to committed objects in the session. The read and write sets of the session are carried forward and continue to accumulate until the session either commits or aborts. Changes made by this session to committed objects are not visible to other sessions until the session commits.

Returns true if accumulated modifications to the committed state of GemStone would not cause concurrency conflict as of the new view; otherwise returns false. If it returns false, you can call the transactionConflicts method to determine the nature of the conflicts.

Warning:
Once `continueTransaction` *has been used within a transaction, a subsequent commit of that transaction will ignore read-write and write-read conflicts. To check for read-write and write-read conflicts, a transaction could use the sequence* `continueTransaction`, `transactionConflicts`, `commitTransaction` *and check the result of* `transactionConflicts` *before doing the* `commitTransaction`.

This method can be used whether or not the session is outside of a transaction. Of course, the session cannot commit the accumulated changes unless it is inside a transaction.

If transaction mode is manualBegin, then `continueTransaction` does not alter the inside/outside of transaction state of the session.

Modifications made by other committed transactions are accumulated for retrieval by GciDirtyObjs() and GciDirtySavedObjs() just as they are accumulated for `commitTransaction` or `abortTransaction`.

This method has no effect on object locks held by the session. Locks in the release locks sets are not released.

`inTransaction` Returns true to indicate that the session is in a transaction, false otherwise.

`transactionMode` Returns the current transaction mode for the current GemStone session, either #autoBegin or #manualBegin. The default is #autoBegin.

`transactionMode:` *newMode*

Sets a new transaction mode for the current GemStone session and exits the previous mode by aborting the current transaction. Valid arguments are #autoBegin and #manualBegin.

Class Protocol

Instance Creation

`currentSession` Returns the sole instance of GsCurrentSession that represents this login session.

`new` Disallowed.

The only instance of GsCurrentSession that is permitted in a session is created automatically when a user logs in to GemStone. Its default SymbolList is a copy of the user's SymbolList. You can obtain that instance by sending the message `GsSession currentSession`.

GsFile

GsFile provides the means for creating and accessing non-GemStone files. Such files reside in the file system on either the machine that is running the current session's Gem process (the server machine) or the machine that is running the client application (the client machine). The files may be of any type, textual or binary, though separate protocol is provided for reading and writing these types of data.

Warning:
Do not retain an instance of GsFile from one session to another. Instances of GsFile are intended to exist only within a given GemStone session. GsFiles that are used across sessions always generate an error.

Superclasses	Object
Named Instance Variables	Intentionally undocumented.
Instance Format	Pointer, Nonindexable, Variant
Subclass Creation	Allowed

Instance Protocol

Accessing

`id`	Returns the stdio 'FILE*' used by the receiver. Returns 0 if no 'FILE*' exists.
`mode`	Returns the access mode of the receiver's file.
`name`	Returns the receiver's file pathname.
`pathName`	Returns the receiver's file path name.

Comparing

`=` *aFile*	Returns true if the receiver and *aFile* represent the same file system file. Returns false otherwise.
`hash`	Returns a SmallInteger related to the value of the receiver. If two instances of GsFile are equal (as compared by the = method), then they must have the same hash value.
`~=` *aFile*	Returns false if the receiver and *aFile* represent the same file system file. Returns true otherwise.

Error Reporting

`lastErrorCode` Returns the currently posted error code, or zero if no error has occurred. Does not clear the error code or error string.

`lastErrorString` Returns the currently posted error string, or nil if no error has occurred. Clears the error string and error code.

File Operations

`close` Closes the receiver's file. Returns the receiver, or nil if an error occurs.

`fileSize` Returns the size in bytes of the receiver, or nil if an error occurs.

Note that the value returned is independent of the open mode used to create the receiver.

`flush` Flushes all written bytes to the file. Returns the receiver, or nil if an error occurs.

`head:` *lineCount* Returns a String containing the first *lineCount* lines from the receiver's file, or nil if an error occurs.

`open` If the receiver is not open, open it using the existing mode. Returns the receiver, or nil if an error occurs.

`open:` *aPathName* `mode:` *openMode*

Opens the receiver's file with the given mode. If the file is already open, it is closed and reopened.

The *openMode* argument must be a String that that is equal to one of the following: `'r'`, `'w'`, `'a'`, `'r+'`, `'w+'`, `'a+'`, `'rb'`, `'wb'`, `'ab'`, `'r+b'`, `'w+b'`, `'a+b'`, `'rb+'`, `'wb+'`, `'ab+'`. The mode has the same meaning as it does for the C library function, fopen().

Returns the receiver if successful, or nil if an error occurs.

Positioning

`atEnd` Returns true if the receiver is currently positioned at the end of its file, false if not, or nil if an error occurs.

`position` Returns the current position of the receiver's file, or nil if an error occurs.

`position:` *offset* Changes the receiver's position in its file by the given offset, which may be negative or zero. Returns the new position, or nil if an error occurs.

`rewind` Repositions the receiver's file to the beginning. Returns 0, or nil if an error occurs.

`seekFromBeginning:` *offset*
Moves the receiver's position in its file to the given offset from the beginning of the file, which may be positive or zero, but not negative. Returns the new position, or nil if an error occurs.

`seekFromCurrent:` *offset*
Changes the receiver's position in its file by the given offset, which may be negative or zero. Returns the new position, or nil if an error occurs.

`seekFromEnd:` *offset* Moves the receiver's position in its file to the given offset from the end of the file, which may be negative or zero, but not positive. Returns the new position, or nil if an error occurs.

Reading

`contents` Returns a String containing the contents of the receiver from the current position to the end of the file. Returns nil if an error occurs.

`next` Returns the next character from the receiver's file, or nil if an error occurs.

`next:` *numberOfBytes* Returns a String containing the next *numberOfBytes* characters from the receiver's file, or nil if an error occurs.

`next:` *amount* `byteStringsInto:` *byteObj*
Reads bytes written by `printBytes:` into the given byte object. Returns count of bytes read, or nil if an error occurs.

`next:` *numberOfBytes* `into:` *aCharacterCollection*

Reads the next *numberOfBytes* into the given collection object. The object's size is truncated to the amount of data actually read. Returns a count of bytes read, or nil if an error occurs.

`next:` *numberOfItems* `ofSize:` *bytesPerItem* `into:` *byteObj*

Reads bytes for the next *numberOfItems* of the given *bytesPerItem* into the given collection object. The object's size is truncated to the amount of data actually read. *bytesPerItem* must between 1 and 4096 inclusive.

Returns a count of bytes read, or nil if an error occurs.

`nextByte` Returns the next byte (integer) from the receiver's file, or nil if an error occurs.

`nextLine` Returns a String containing the next line from the receiver's file. The String will be terminated with a newline, unless the end of file is reached and there is no line terminator. If the receiver is positioned at the end of the file, returns an empty String. Returns nil if an error occurs.

If the file contains binary data including NULL characters, it may not be possible to read beyond the NULL character. `GsFile | next:ofSize:into:` should be used to read binary files.

`nextLineInto:` *str* `startingAt:` *pos*

Reads the next line from the receiver's file into the given collection object, starting at the given position in the collection. The collection will be terminated with a newline, unless the end of file is reached and there is no line terminator. If the receiver is positioned at the end of the file, nothing is written. Returns a count of bytes read, or nil if an error occurs.

If the file contains binary data including NULL characters, it may not be possible to read beyond the NULL character. `GsFile | next:ofSize:into:` should be used to read binary files.

`peek`	Returns the next byte in the receiver's file, without advancing the current pointer. Returns nil if an error occurs.
`peek2`	Returns the next byte plus one in the receiver's file, without advancing the current pointer. Returns nil if an error occurs.
`skip:` *count*	Changes the receiver's position in its file by the given offset, which may be zero, but not negative. Returns the new position, or nil if an error occurs.

Testing

`isClient`	Returns true if the receiver's file is a client file, or nil if an error occurs.
`isExternal`	Is the source for this stream is external to GemStone Smalltalk.
`isOpen`	Returns true if the receiver's file is open, or nil if an error occurs.

Writing

`+` *collection*	Writes the contents of the given collection to the receiver's file at the end of the file. The argument must be a kind of CharacterCollection with byte format. Returns a count of bytes added, or nil if an error occurs.
`add:` *char*	Writes the given Character to the receiver's file at the end of the file. Returns true, or nil if an error occurs.
`addAll:` *collection*	Writes the contents of the given collection to the receiver's file at the end of the file. The argument must be a kind of CharacterCollection with byte format. Returns a count of bytes added, or nil if an error occurs.
`cr`	Writes a carriage return to the receiver's file. Returns a count of bytes added, or nil if an error occurs.
`ff`	Writes a form-feed (page break) to the receiver's file. Returns true, or nil if an error occurs.
`lf`	Writes a line-feed to the receiver's file. Returns a count of bytes added, or nil if an error occurs.

`log:` *string*
Writes the contents of the given collection to the receiver's file at the current position. Appends a newline to the file if the string does not end with one. The argument must be a kind of CharacterCollection with byte format.

Returns the receiver if successful; returns nil otherwise.

`nextPut:` *aByte*
Writes the given byte to the receiver's file at the current position. *aByte* must be a Character or a SmallInteger in the range 0..255.

If *aByte* is a SmallInteger, it will be interpreted logically as

Character withValue: *aByte*

Returns true, or nil if an error occurs.

`nextPutAll:` *collection*
Writes the contents of the given collection to the receiver's file at the current position. The argument must be a kind of CharacterCollection with byte format. Returns a count of bytes added, or nil if an error occurs.

`nextPutAllBytes:` *collection*
Writes the byte contents of the given collection to the receiver's file at the current position. The argument must be a kind of CharacterCollection with byte format. Returns a count of bytes added, or nil if an error occurs.

`printBytes:` *byteObj*
Prints the bytes from the given byte object in decimal notation with line breaks to keep output lines from being too long.

Returns the receiver, or nil if an error occurs.

See also the method `next:byteStringsInto:`.

`write:` *byteObj* `itemCount:` *itemCount* `ofSize:` *bytesPerItem*
Writes *itemCount* items of size *bytesPerItem* from the given byte object to the receiver's file. *bytesPerItem* must be between 1 and 4096 inclusive. Returns a count of bytes written, or nil if an error occurs.

Class Protocol

Directory Operations

`contentsAndTypesOfDirectory:` *dirSpec* `onClient:` *bool*

Returns an Array of objects describing the contents of the given directory. The location of the directory is indicated by *bool*. Successive pairs of elements of the Array will be the name of an entry, and a Boolean - true if the entry is a file, and false if not.

Sample: #[`file.c, true, subdir, false, ...`]

Returns anArray if successful, nil if not.

`contentsOfDirectory:` *dirSpec* `onClient:` *bool*

Returns an Array of Strings describing the contents of the given directory. The location of the directory is indicated by *bool*. Each element of the Array will be the name of an entry.

Returns anArray if successful, nil if not.

`exists:` *aPathName*

Returns true if the given path points to a file on the client, false if not, and nil if an error occurs trying to find out.

`existsOnServer:` *aPathName*

Returns true if the given path points to a file, on the server, false if not, and nil if an error occurs trying to find out.

`sizeOf:` *aPathName*

Returns the size in bytes of the given client file, or nil if an error occurs.

`sizeOfOnServer:` *aPathName*

Returns the size in bytes of the given server file, or nil if an error occurs.

Error Reporting

`lastErrorString`

Returns the currently posted error string, for class operations on the client, or nil if no error has occured. Clears the error string.

`serverErrorString`

Returns the currently posted error string, for class operations on the server, or nil if no error has occured. Clears the error string.

File Operations

`closeAll`
Closes all open files on the client machine except stdin/stdout/stderr. Returns the receiver if successful, nil if not.

`closeAllOnServer`
Closes all open files on the server machine except stdin/stdout/stderr. Returns the receiver if successful, nil if not.

`removeClientFile:` *aPathName*
Removes the named file from the client machine's file system. Returns the receiver if the file was deleted, nil if an error occurs.

`removeServerFile:` *aPathName*
Removes the named file from the server machine. Returns the receiver if the file was deleted, nil if an error occurs.

Instance Creation

`newWithFilePtr:` *filePtr* `pathname:` *aPathName* `mode:` *openMode* `onClient:` *clientBool*
Creates an instance of the receiver using an already open FILE*: *filePtr*. *aPathName*, *openMode*, and *clientBool* describe the already open *filePtr*. The *filePtr* most likely comes from a C user action. This method assumes it was converted to an object using GciLongToOop().

Returns a GsFile if successful, nil if an error occurs.

`open:` *aPathName* `mode:` *openMode*
Creates an instance of the receiver and opens a file on the client machine.

The *openMode* argument must be a String that that is equal to one of the following: `'r'`, `'w'`, `'a'`, `'r+'`, `'w+'`, `'a+'`, `'rb'`, `'wb'`, `'ab'`, `'r+b'`, `'w+b'`, `'a+b'`, `'rb+'`, `'wb+'`, `'ab+'`. The mode has the same meaning as it does for the C library function, fopen().

Returns a GsFile if successful, nil if an error occurs.

`open:` *aPathName* `mode:` *openMode* `onClient:` *clientBool*

Creates an instance of the receiver and opens a file on the client machine (if *clientBool* is true) or the server machine (if *clientBool* is false).

The *openMode* argument must be a String that that is equal to one of the following: `'r'`, `'w'`, `'a'`, `'r+'`, `'w+'`, `'a+'`, `'rb'`, `'wb'`, `'ab'`, `'r+b'`, `'w+b'`, `'a+b'`, `'rb+'`, `'wb+'`, `'ab+'`. The mode has the same meaning as it does for the C library function, fopen().

Returns a GsFile if successful, nil if an error occurs.

`openAppend:` *aPathName*

Opens a file on the client machine. The file is opened for append. The file is treated as a text file. Returns aGsFile if successful, nil if an error occurs.

`openAppendOnServer:` *aPathName*

Opens a file on the server machine. The file is opened for append. The file is treated as a text file. Returns aGsFile if successful, nil if an error occurs.

`openOnServer:` *aPathName* `mode:` *openMode*

Creates an instance of the receiver and opens a file on the server machine.

The *openMode* argument must be a String that that is equal to one of the following: `'r'`, `'w'`, `'a'`, `'r+'`, `'w+'`, `'a+'`, `'rb'`, `'wb'`, `'ab'`, `'r+b'`, `'w+b'`, `'a+b'`, `'rb+'`, `'wb+'`, `'ab+'`. The mode has the same meaning as it does for the C library function, fopen().

Returns a GsFile if successful, nil if an error occurs.

`openRead:` *aPathName* Opens a file on the client machine. The file is opened for read. The file is treated as a text file. Returns aGsFile if successful, nil if an error occurs.

`openReadOnServer:` *aPathName*

Opens a file on the server machine. The file is opened for read. The file is treated as a text file. Returns aGsFile if successful, nil if an error occurs.

`openWrite:` *aPathName* Opens a file on the client machine. The file is opened for write. The file is treated as a text file. Returns aGsFile if successful, nil if an error occurs.

`openWriteOnServer:` *aPathName*
Opens a file on the server machine. The file is opened for write. The file is treated as a text file. Returns aGsFile if successful, nil if an error occurs.

Standard Files

`stderr` Returns an instance of the receiver that is set up to write to the standard error output of the client process, or nil if an error occurs.

`stdin` Returns an instance of the receiver that is set up to read the standard input of the client process, or nil if an error occurs.

`stdout` Returns an instance of the receiver that is set up to write to the standard output of the client process, or nil if an error occurs.

GsInterSessionSignal

A GsInterSessionSignal represents a signal from one session to another within a single GemStone system.

Superclasses Object

Named Instance Variables

sessionSerialNum — A SmallInteger identifier of the session from which the instance was received. To obtain the corresponding session, use the method `GsSession | sessionWithSerialNumber:`.

signal — A SmallInteger representing application-defined information from the sending session.

message — A String representing application-defined information from the sending session.

Instance Format Pointer, Nonindexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`message`	Returns the String sent as a message.
`message:` *aString*	Sets the String to be sent as a message.
`session`	Returns a transient GsSession object representing the session that sent the signal, or nil if there was no signal. This object can be used as target of signals sent in response.
`session:` *aGsSession*	Sets the **session** instance variable so that it represents the session that sent the signal.
`signal`	Returns the SmallInteger sent as a signal.
`signal:` *aSmallInteger*	Sets the SmallInteger to be sent as a signal.

Signalling

`replyToSenderWithSignal:` *aSmallInteger* `withString:` *aString*

Sends a signal containing the arguments to the originating session of the receiver. If the **session** instance variable of the receiver is nil, raises an error.

`sendToSession:` *aGsSession*

Sends a signal to the session represented by *aGsSession*. The signal contains the **signal** and **message** instance variables of the receiver. Ignores the **session** instance variable of the receiver.

Class Protocol

Instance Creation

`signal:` *aSignal* `message:` *aString*

Returns a new instance with the given information installed. The originating GsSession is set to nil.

GsMethod

A GsMethod is a compiled form of a GemStone Smalltalk method.

Superclasses Object

Named Instance Variables

invocationCount — A SmallInteger count of the number of times the method is currently active on the stack.

numBreakpoints — A SmallInteger count of breakpoints defined on the method.

selector — A Symbol that defines the method's selector.

literalsOffset — A SmallInteger that gives the index where literals are stored in an instance.

numArgs — A SmallInteger that defines the number of arguments that the method expects.

inClass — The Behavior (a Class or Metaclass) for which the method was compiled.

numSends — For GemStone internal use.

sourceString — A CharacterCollection containing the source code of the method.

debugInfo — An Array that captures information that is useful in debugging.

Instance Format Pointer, Indexable, Variant

Subclass Creation Disallowed

Instance Protocol

Accessing

`argsAndTemps` Returns an Array of Symbols which are the names of arguments and temporaries for this method.

`inClass` Returns the class in which the receiver was compiled.

`invocationCount` Returns the value of the instance variable named **invocationCount**.

`literals` Returns an Array containing the literal pool of the receiver.

`literalsOffset` Returns the value of the instance variable named **literalsOffset**.

`numArgs` Returns the value of the instance variable named **numArgs**.

`selector` Returns the value of the instance variable named **selector**.

`sourceString` Returns a CharacterCollection that contains the source code of the receiver.

Clustering

`clusterDepthFirst` This method clusters the receiver, its bytecodes, its selector pool, and its **selector** in depth-first order. Returns true if the receiver has already been clustered during the current transaction; returns false otherwise.

Copying

`copy` Disallowed. You may not create new instances of GsMethod.

Debugging Support

`clearAllBreaks` Clear all method breakpoints in the receiver.

`clearBreakAtStepPoint:` *aStepPoint*
Clear method breakpoint at specified step point.

`disableAllBreaks` Disable all method breakpoints in the receiver.

`disableBreakAtStepPoint:` *aStepPoint*
Disable method breakpoint at specified step point.

`setBreakAtStepPoint:` *aStepPoint*
Set method breakpoint at specified step point.

Decompiling without Sources

`decompileForCategory:` *aCategory* `classRef:` *classRefExpression*
`stripWith:` *sourceStripSelector* `classMethod:` *isMeta*
Decompiles the receiver to produce a Topaz run command that will regenerate it.

Reporting

`isSenderOf:` *aSymbol* Returns true if the receiver sends the message *aSymbol*. Returns false otherwise.

Storing and Loading

`writeTo:` *aPassiveObject* — Instances of GsMethod cannot be converted to passive form. This method writes nil to *aPassiveObject* and stops GemStone Smalltalk execution with a notifier.

Stripping Sources

`emptySource` — Returns nil in place of the source string. The `emptySource` selector may be used as an argument to the `stripWith:` keyword of the method `GsMethod>>decompileForCategory:classRef:stripWith:classMethod:`, where it causes the string 'source not available...' to be used as the source string when reloading the decompiled method.

`fullSource` — Returns the complete source string. The `fullSource` selector may be used as an argument to the `stripWith:` keyword of the method `GsMethod>>decompileForCategory:classRef:stripWith:classMethod:`.

`removeAllSourceButFirstComment` — Installs a new source string for the receiver so that only the method signature and the first comment (if it exists) are left. For use in stripping a method in place in GemStone. Bypasses the invariance of the receiver, but still requires write authorization to the Segment of the receiver.

`sourceToFirstComment` — Returns a new source string for the receiver that contains only the method signature and the first comment (if it exists). Does not modify the receiver. The `sourceToFirstComment` selector may be used as an argument to the `stripWith:` keyword of the method `GsMethod>>decompileForCategory:classRef:stripWith:classMethod:`.

Class Protocol

Debugging Support

`clearAllBreaks` Clear all method breakpoints that have been set in any methods.

`clearBreakInClass:` *aClass* `selector:` *aSelector* `stepPoint:` *aStepPoint*
Clear the breakpoint at *aStepPoint* in method *aSelector* of class *aClass*.

`disableBreakInClass:` *aClass* `selector:` *aSelector* `stepPoint:` *aStepPoint*
Disable the breakpoint previously set at *aStepPoint* in method *aSelector* of class *aClass*.

`enableBreakInClass:` *aClass* `selector:` *aSelector* `stepPoint:` *aStepPoint*
Set or reenable the breakpoint previously set at *aStepPoint* in method *aSelector* of class *aClass*.

Instance Creation

`new` Disallowed. You cannot create new instances of GsMethod.

`new:` *anInteger* Disallowed. You cannot create new instances of GsMethod.

GsMethodDictionary

GsMethodDictionary optimizes IdentityKeyValueDictionary for use as method dictionaries in classes. It employs a different internal structure that is well-suited for efficient execution in smaller dictionaries.

The keys of method dictionaries must be canonical symbols (Symbols or DoubleByteSymbols).

Superclasses IdentityKeyValueDictionary, KeyValueDictionary, AbstractDictionary, Collection, Object

Named Instance Variables **valueConstraint —** The Class that specifies a constraint on the dictionary's values. If nil, there is no constraint.

keyConstraint — The Class that specifies a constraint on the dictionary's keys. If nil, there is no constraint.

Instance Format Pointer, Indexable, Variant

Subclass Creation Disallowed

Instance Protocol

Accessing

`at:` *aKey* `ifAbsent:` *aBlock*

Returns the value whose key is identical to *aKey*. If no such key/value pair exists, returns the result of evaluating the zero-argument block *aBlock*.

`at:` *aKey* `otherwise:` *aValue*

Returns the value whose key is identical to *aKey*. If no such key/value pair exists, returns the given alternate value.

`keyAtValue:` *anObject* `ifAbsent:` *aBlock*

Returns the key of the first value identical to *anObject*. If no match is found, this method evaluates the block *aBlock* and returns its result.

`keyConstraint` Returns the key constraint of the receiver.

`valueConstraint` Returns the value constraint of the receiver.

Clustering

`clusterDepthFirst` This method clusters the receiver and its values in depth-first order. The keys are not clustered because they are Symbols.

Has no effect and returns true if the receiver was previously clustered in the current transaction.

Copying

`copy` Returns a copy of the receiver which shares the receiver's instance variables.

Enumerating

`associationsDo:` *aBlock*

Evaluates *aBlock* with each of the receiver's key/value pairs as the argument by creating a SymbolAssociation for each key/value pair. The argument *aBlock* must be a one-argument block. Returns the receiver.

`keysAndValuesDo:` *aBlock*

Evaluates *aBlock* with each of the receiver's key/value pairs as the arguments. The argument *aBlock* must be a two-argument block. The first argument is the key and the second argument is the value of each key/value pair. Returns the receiver.

Formatting

`printOn:` *aStream* Puts a displayable representation of the receiver on the given stream.

Hashing

`hashFunction:` *aKey* The hash function performs an operation on the value of the key *aKey* and returns some Integer between 1 and tableSize, inclusive.

`rebuildTable:` *newSize* Rebuilds the method dictionary by populating a larger method dictionary first and doing a (primitive) become:

Initializing

`initialize:` *newSize* Initializes the instance variables of the receiver to be an empty IdentityKeyValueDictionary of the specified size.

Removing

`removeAll` Remove all key/value pairs from the receiver.

`removeKey:` *aKey* `ifAbsent:` *aBlock*
Removes the key/value pair whose key is identical to *aKey*. If no such key/value pair exists, returns the result of evaluating the zero-argument block *aBlock*.

Statistics

`statistics` A GsMethodDictionary has no collision buckets, so the statistics defined for KeyValueDictionary have no meaning.

Updating

`at:` *aKey* `put:` *aValue* Stores the aKey/aValue pair in the hash dictionary. Rebuilds the hash table if the addition caused the number of collisions to exceed the limit allowed.

If *aKey* is not compatible with the key constraint of the receiver, or *aValue* is not compatible with the value constraint of the receiver, an error is generated.

`changeToSegment:` *segment*
Assigns the receiver to the given segment.

`keyConstraint:` *aClass* Sets the key constraint of the receiver to *aClass*. Generates an error if the receiver is not empty.

`valueConstraint:` *aClass*
Sets the value constraint of the receiver to *aClass*. Generates an error if the receiver is not empty.

GsProcess

A GsProcess represents a suspended GemStone Smalltalk call stack, including information needed to restart execution.

Superclasses Object

Named Instance Variables

stackDepth — A positive SmallInteger, the number of active methods on the stack of the GsProcess.

controlStack — A GsStackBuffer, the saved control stack.

arStack — A GsStackBuffer, the saved evaluation stack.

inUserActionCount — A SmallInteger, for GemStone internal use.

interruptFlag — A SmallInteger, for GemStone internal use.

fltStatus — A String, for GemStone internal use.

recursionsToStCount — A SmallInteger, for GemStone internal use.

protectedMode — A SmallInteger, for GemStone internal use.

asyncEventsDisabled — A Boolean, true if asynchronous events are disabled, false if they are enabled.

Instance Format Pointer, Nonindexable, Variant

Subclass Creation Disallowed

Instance Protocol

Accessing

`methodAt:` *aLevel* Returns the GsMethod that is active at *aLevel* in the receiver, where *aLevel* == 1 is the top of the stack. Generates an error if *aLevel* less than zero or greater than **stackDepth**. Returns nil if there is a reenter marker at the specified level.

`stackDepth` Returns the value of the **stackDepth** instance variable.

Copying

`copy` Disallowed.

Formatting

`printOn:` *aStream* Puts a displayable representation of the receiver on the given stream.

`printString` Returns a String whose contents are a displayable representation of the receiver.

Updating

`instVarAt:` *anIndex* `put:` *aValue*

Disallowed.

Class Protocol

Debugging Support

`stackReportToLevel:` *aLevel*

Returns a String describing the currently active stack, starting with to the sender of this method (which is considered level 1). The *aLevel* argument specifies the depth to which to report the stack.

The format of the result is subject to change with each release of GemStone.

Instance Creation

`new` Disallowed.

GsSession

A GsSession represents a user session on the GemStone server where the instance exists. It is a transient object that is useful only as long as the user remains logged in to GemStone.

A GsSession can access the UserProfile of the user who is logged in to GemStone, can provide some minimal control over session execution, and can send and receive signals with other sessions.

The current session is represented by a GsCurrentSession, a special subclass of GsSession. Instances of GsSession typically represent other concurrent GemStone sessions in the same server.

Superclasses AbstractSession, Object

Named Instance Variables

sessionSerialNum — A SmallInteger that identifies the session uniquely within the GemStone server.

userProfile — The UserProfile of the user who is logged in to the session.

Instance Format Pointer, Nonindexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`remoteSessions` Returns a collection of GsRemoteSession instances representing all the remote sessions spawned by the session represented by the receiver. If the receiver has no remote sessions, the resulting collection is empty. This list is maintained in the session transient state.

`serialNumber` Returns the serial number issued to the session represented by the receiver when that session logged in.

`sessionSerialNum` Returns the value of the **sessionSerialNum** instance variable of the reciever.

`userProfile` Returns the UserProfile attached to the session.

Session Control

`stop` Aborts the current transaction and terminates the session. If the receiver is the current session, no operation is performed. If the UserProfile of the current session lacks SystemControl privilege, an exception occurs.

Signalling

`enableInterSessionSignalling:` *aBoolean*

If *aBoolean* is true, enables receipt of intersession signals in the session represented by the receiver. Otherwise, disables receipt of such signals.

`sendSignalObject:` *aGsInterSessionSignal*

Sends the signal and message of *aGsInterSessionSignal* to the session represented by the receiver.

Testing

`hasRemoteSessions` Returns true if the receiver has remote sessions, false otherwise.

If the receiver is itself a remote session, returns nil.

`isCurrent` Returns true if the receiver represents the current login session of this Gem; returns false otherwise.

`isRemote` Returns true if the receiver represents a remote federated session.

Class Protocol

Instance Creation

`currentSession` Returns the sole instance of GsCurrentSession that represents this login session.

`sessionWithSerialNumber:` *anInteger*

Returns an instance of a kind of GsSession that represents the GemStone session on the same repository, whose serial number is *anInteger*. Returns nil if no logged-in session has that serial number.

Requires SessionAccess privilege if the session described by *anInteger* is not the current session.

GsSocket

GsSocket provides the means for creating and binding TCP sockets through the operating system of the machine that is running the session's Gem process, and for communicating across those sockets. Methods that block GemStone Smalltalk until the socket operation completes are interruptable by a hard break. (You can get a hard break in Topaz by pressing the control-C key twice. You can get a hard break in GemBuilder for C by calling the GciHardBreak function, and in GemBuilder for Smalltalk by calling the corresponding hard break method.)

Warning:
Do not retain an instance of GsSocket from one session to another. Instances of GsSocket are intended to exist only within a given GemStone session. GsSockets that are used across sessions always generate an error.

Superclasses	Object
Named Instance Variables	None
Instance Format	Pointer, Nonindexable, Variant
Subclass Creation	Allowed

Instance Protocol

Accessing

`id`	Returns the value of the low level socket. If no low level socket exists, returns -1.
`peerName`	For a bound socket, returns the hostname of the machine on which the process at the other end of the connection is running. If the socket is not bound, or an error occurs, returns nil.
`port`	Returns the port number of a bound socket, or nil if not bound or an error occurs.

Backward Compatibility

Methods in this category are obsolete and are provided only for compatibility with earlier releases of GemStone. They will be removed in a future release.

`bind`	Obsolete in GemStone 5.0. Use the `bindTo:` method instead.

`listen:` *queueLength* Obsolete in GemStone 5.0.

`listen:` *queueLength* `acceptingWith:` *aSocket*
Obsolete in GemStone 5.0.

`readReady` Obsolete in GemStone 5.0. Use the `readWillNotBlock` or `readWillNotBlockWithin:` method instead.

`writeReady` Obsolete in GemStone 5.0. Use the `writeWillNotBlock` or `writeWillNotBlockWithin:` method instead.

Client Operations

`bindTo:` *portNumber* Binds the receiver to the specified port number. Use the `makeServer` methods to do the bind when creating a server socket. This method is provided to bind a client socket to a specific port before it is connected. Returns the port number actually bound to (should be the same as the argument unless argument is nil), or nil if not successful.

`connectTo:` *portNumber* `on:` *hostName*
Connect the receiver to the server socket identified by *portNumber* and *hostName*. Returns true if the connection succeeded and false if not.

Comparing

`=` *aSocket* Returns true if the receiver and *aSocket* represent the same operating system socket. Returns false otherwise.

`hash` Returns a SmallInteger related to the value of the receiver. If two instances of GsSocket are equal (as compared by the = method), then they must have the same hash value.

`~=` *aSocket* Returns false if the receiver and *aSocket* represent the same operating system socket. Returns true otherwise.

Error Reporting

`lastErrorCode` Returns an integer representing that last operating system error on the receiver. Returns zero if there is no error. Does not clear the error code or string.

`lastErrorString` Returns the string of the last error on the receiver, or nil if no error has occurred. Clears the error code and string.

Reading

`read:` *maxBytes*
This method is equivalent to `readString:`.

`read:` *maxBytes* `into:` *byteObj*
Reads up to the given number of bytes into the given byte object (for example, a String). Returns the number of bytes read, or nil if an error occurs.

If no data are available for reading, this blocks until data arrives. The `readWillNotBlock` or `readWillNotBlockWithin:` methods may be used to determine whether or not data is ready for reading before calling this method.

`readString:` *maxBytes*
Reads up to the given number of bytes, returning them in a String whose size is between 1 and *maxBytes* inclusive. If an error occurs, nil is returned instead.

If *maxBytes* is greater than the size of the operating system's buffer for the socket, the size of the result string may be a function of this buffer size, even if more data is available from the sender. Repeated invocation of this method may be necessary to obtain all of the data.

For optimium performance, *maxBytes* should be <= 8192, and should typically be 4096.

If no data are available for reading, this blocks until data arrives. The `readWillNotBlock` or `readWillNotBlockWithin:` methods may be used to determine whether or not data is ready for reading before calling this method.

Server Operations

`accept`
Accept a client request for a connection on the receiver. Returns the socket created for a new connection, or nil if there was some problem. For example, the following code does not return until there is a connection:

```
sock := GsSocket new.
sock makeServer.
newsock := sock accept.
msg := newsock read: 512.
```

`makeServer` Turns the receiver into a server socket. Binds the receiver to a port and sets makes it a listening socket. Returns the receiver or nil if an error occurred.

`makeServer:` *queueLength*

Turns the receiver into a server socket. The *queueLength* specified the size of the listen backlog queue for incoming connections. Binds the receiver to a random port. Returns the receiver or nil if an error occurred.

`makeServer:` *queueLength* `atPort:` *portNum*

Turns the receiver into a server socket. The *queueLength* specified the size of the listen backlog queue for incoming connections. Binds the receiver to *portNum*. If *portNum* is nil then a random port is selected. Returns the receiver, or nil if an error occurred.

`makeServerAtPort:` *portNum*

Turns the receiver into a server socket. Binds the receiver to *portNum* and makes it a listening socket. Returns the receiver or nil if an error occurred.

Socket Operations

`close` Release any temporary system resources used by the receiver. This includes closing the low level socket. Returns self if the socket is closes successfully or the socket is already closed. Returns nil if socket cannot be closed.

`keepAlive:` *bool* Sets the receiver to periodically broadcast messages to clients. If a client does not respond to a broadcast, its connection is severed. If the parameter, *bool*, is false, keepAlive is turned off. Returns the receiver or nil if an error occured.

`linger:` *bool* `length:` *timeOut*

Sets up the receiver so that if unsent data is waiting to be transmitted at the time the receiver is closed, the current process will block until either the data is transmitted, or the given *timeOut* expires. *timeOut* is in units of seconds.

Returns the receiver or nil if an error occured.

Testing

`readWillNotBlock`

Returns true if the socket is currently ready to receive input without blocking. Returns false if it is not currently ready. Returns nil if an error occurs.

The receiver must already be connected for this method to work properly. If it is not connected, then the value that this method returns is indeterminate. Use the `peerName` method to determine if the receiver is connected.

Call this method to prevent subsequent read or accept operations from hanging. If it returns true for a connected socket, then the input operation will not hang. However, a return value of true is no guarantee that the operation itself will succeed.

`readWillNotBlockWithin:` *msToWait*

Returns true if the socket is ready to receive input without blocking within *msToWait* milliseconds from the time that this method is called. Returns false if it is not ready after *msToWait* milliseconds. Returns nil if an error occurs.

If *msToWait* is 0, then this method reports the current readiness of the receiver. If *msToWait* is -1, then this method never returns false, but waits until the receiver is ready to receive input without blocking, and then returns true.

The receiver must already be connected for this method to work properly. If it is not connected, then the value that this method returns is indeterminate. Use the `peerName` method to determine if the receiver is connected.

Call this method to prevent subsequent read or accept operations from hanging. If it returns true for a connected socket, then the input operation will not hang. However, a return value of true is no guarantee that the operation itself will succeed.

`writeWillNotBlock`

Returns true if the socket is currently ready to take output without blocking. Returns false if it is not currently ready. Returns nil if an error occurs.

The receiver must already be connected for this method to work properly. If it is not connected, then the value that this method returns is indeterminate. Use the `peerName` method to determine if the receiver is connected.

Call this method to prevent subsequent write operations from hanging. If it returns true for a connected socket, then a subsequent write will not hang. However, a return value of true is no guarantee that the write operation itself will succeed.

`writeWillNotBlockWithin:` *msToWait*

Returns true if the socket is ready to take output without blocking within *msToWait* milliseconds from the time that this method is called. Returns false if it is not ready after *msToWait* milliseconds. Returns nil if an error occurs.

If *msToWait* is 0, then this method reports the current readiness of the receiver. If *msToWait* is -1, then this method never returns false, but waits until the receiver is ready to take output without blocking, and then returns true.

The receiver must already be connected for this method to work properly. If it is not connected, then the value that this method returns is indeterminate. Use the `peerName` method to determine if the receiver is connected.

Call this method to prevent subsequent write operations from hanging. If it returns true for a connected socket, then a subsequent write will not hang. However, a return value of true is no guarantee that the write operation itself will succeed.

Writing

`write:` *byteObj*
Write out the given byte object. Returns the number of bytes written, or nil if an error occurs.

If the stream is not ready for writing, this blocks until it is ready. The `writeWillNotBlock` or `writeWillNotBlockWithin:` methods may be used to determine whether or not data is ready for writing before calling this method.

`write:` *amount* `from:` *byteObj*
Write the given number of bytes from the given byte object. Returns the number of bytes written, or nil if an error occurs.

If the stream is not ready for writing, this blocks until it is ready. The `writeWillNotBlock` or `writeWillNotBlockWithin:` methods may be used to determine whether or not data is ready for writing before calling this method.

Class Protocol

Drastic Measures

`closeAll`
Close all instances of GsSocket that are open.

Error Reporting

`lastErrorCode`
Returns an integer representing that last operating system error for GsSocket class methods. Returns zero if there is no error. Does not clear the error code or string.

`lastErrorString`
Returns a String containing information about the last error for GsSocket class methods, or nil if there is no error.

Clears the last error string and number.

Examples

`clientExample` — This client will connect to a server created with `serverExample`, and read the string object that the server sends.

Creates a socket, connect it to port 57785, read a string, close the socket, and check the resulting object. Returns true if successful.

The server should already be listening for connections when this method is invoked.

`serverExample` — Creates a socket, binds it to port 57785, and waits for a connection. When a connection is established, sends some data to the client, and closes the connection. Returns true if successful.

You will need two GemStone sessions running from two independent interface processes to run both this and the `clientExample`. The gem processes for the two sessions must be on the same machine. (For example two Topaz sessions.)

Warning: This method will cause your current session to hang until a connection is established.

Instance Creation

`basicNew` — Creates a new, uninitialized instance of the receiver.

`new` — Returns a new socket or nil if unable to create a new socket.

Queries

`getServByName:` *serviceName*

Returns the port number for the given service. Returns nil if the service is undefined or an error occurs.

`isAvailable` — Returns whether the supporting socket actions are available in the user's session.

IdentityBag

An IdentityBag is an UnorderedCollection in which any distinct object can occur any number of times. Adding the same (identical) object to an IdentityBag multiple times simply causes it to occur multiple times in the IdentityBag.

Since an IdentityBag is an identity-based collection, different (non-identical) but equivalent (equal) objects are treated as distinct from each other. In Bags, they are not distinct. Adding multiple equivalent objects to an IdentityBag yields an IdentityBag with multiple objects as elements, each occurring once.

You can create subclasses of IdentityBag to restrict the kind of elements it contains. When creating a subclass of IdentityBag, you must specify a class as the aConstraint argument. This class is called the element kind of the new subclass. For each instance of the new subclass, the class of each element must be of the element kind.

Superclasses	UnorderedCollection, Collection, Object
Named Instance Variables	None
Instance Format	Nsc, Nonindexable, Variant
Subclass Creation	Allowed

Instance Protocol

Accessing

`at:` *anIndex*

Returns the element of the receiver that is currently located at position *anIndex*.

The elements of an IdentityBag are inherently unordered, and can change position (index) when the IdentityBag is altered. Thus, after an IdentityBag is altered, a given element may reside at a different index than before, and a given index may house a different element. You should not infer an ordering for an IdentityBag's elements when you access them by index.

This method is useful primarily as a code optimizer for iterating over all the elements of an IdentityBag (using a loop that runs the index from 1 to the size of the IdentityBag).

The IdentityBag must not change during the iteration. But the iteration may run faster than it would if you use other alternatives such as the `do:` method.

`instVarAt:` *aSmallInteger*

If the variable has a publicly accessible named instance variable at index *aSmallInteger*, this returns its value. Generates an error if *aSmallInteger* is not a SmallInteger or is out of bounds, or if the receiver has no publicly accessible named instance variables.

`instVarAt:` *anIndex* `put:` *aValue*

Stores the argument *aValue* in the instance variable indicated by *anIndex* and returns *aValue*. Generates an error if (1) *anIndex* is not a SmallInteger, (2) *anIndex* is out of bounds or (3) if the receiver has no publicly accessible named instance variables.

Adding

`add:` *newObject*
Adds *newObject* to the receiver. Has no effect if *newObject* is nil.

`add:` *anObject* `withOccurrences:` *anInteger*
Includes *anObject* as an element of the receiver *anInteger* number of times. Generates an error if *anObject* is not a kind of the bag's element kind. Has no effect if *anObject* is nil.

`addAll:` *aCollection*
Adds all of the elements of *aCollection* to the receiver. If *aCollection* is a kind of KeyValueDictionary, then this method adds new Associations that reference the key/value pairs found in *aCollection*.

Comparing

`=` *aBag*
Returns true if all of the following conditions are true:

1. The receiver and *aBag* are of the same class.
2. The two collections are of the same size.
3. They have the same element kind.
4. Their public named instance variables are identical.
5. The elements of the receiver and *aBag* are identical.
6. Each element occurs the same number of times in the receiver and in *aBag*.

Returns false otherwise.

`hash`
Returns an Integer hash code for the receiver.

Enumerating

`do:` *aBlock*
Evaluates *aBlock* with each of the receiver's elements as the argument. The argument *aBlock* must be a one-argument block.

Instance Migration

`migrateFrom:` *anotherObject* `instVarMap:` *otherivi*

Takes information from the given object and puts it in the receiver. This message is sent to an object when its class is being migrated to another class to account for changes in a schema. The *otherivi* argument is a precalculated indirection table associating the receiver's instance variables with instance variables in the other object. If a table entry is 0, the other object is assumed not to have that instance variable.

This method should be augmented to perform other necessary initializations in the receiver.

Removing

`remove:` *anObject* Removes *anObject* from the receiver and returns *anObject*. If *anObject* is present several times in the receiver, only one occurrence is removed. Generates an error if *anObject* is not in the receiver.

`remove:` *anObject* `ifAbsent:` *exceptionBlock*

Removes from the receiver an object that is identical to *anObject* and returns *anObject*. If several elements of the receiver are identical to oldObject, only one instance is removed. If oldObject is not present in the receiver, evaluates anExceptionBlock and returns the result of the evaluation.

`removeAll:` *aCollection* Removes one occurrence of each element of *aCollection* from the receiver and returns the receiver. Generates an error if any element of *aCollection* is not present in the receiver.

`removeAllPresent:` *aCollection*

Removes from the receiver one occurrence of each element of *aCollection* that is also an element of the receiver. Differs from `removeAll:` in that, if some elements of *aCollection* are not present in the receiver, no error is generated. Returns *aCollection*.

`removeIdentical:` *anObject*

Removes *anObject* from the receiver and returns *anObject*. If *anObject* is present several times in the receiver, only one occurrence is removed. Generates an error if *anObject* is not in the receiver.

`removeIdentical:` *anObject* `ifAbsent:` *exceptionBlock*

Removes from the receiver an object that is identical to *anObject* and returns *anObject*. If several elements of the receiver are identical to oldObject, only one instance is removed. If oldObject is not present in the receiver, evaluates anExceptionBlock and returns the result of the evaluation.

`removeIfPresent:` *anObject*

Removes *anObject* from the receiver and returns *anObject*. If *anObject* is present several times in the receiver, only one occurrence is removed. Returns nil if *anObject* is missing from the receiver.

Searching

`collect:` *aBlock*

Evaluates *aBlock* with each of the receiver's elements as the argument. Collects the resulting values into a collection of the same class as the receiver, and returns the new collection. The argument *aBlock* must be a one-argument block.

`includes:` *anObject*

Returns true if the argument *anObject* is an element of the receiver. Returns false otherwise. (Compare with `includesValue:`, which is based on equality.)

`includesIdentical:` *anObject*

Returns true if the argument *anObject* is an element of the receiver. Returns false otherwise. (Compare with `includesValue:`, which is based on equality.)

`includesValue:` *anObject*

Returns true if the receiver contains an object of the same value as the argument, *anObject*. Returns false otherwise. (Compare with `includes:`, which is based on identity.)

`occurrencesOf:` *anObject*

Returns the number of the receiver's elements that are identical (==) to *anObject*.

`speciesForCollect` Returns a class, an instance of which should be used as the result of `collect:` or other projections applied to the receiver.

Set Arithmetic

`*` *aBagOrSet* Intersection. Returns a kind of IdentityBag containing only the elements that are present in both the receiver and the argument *aBagOrSet*.

The class of the result is the lowest class in the hierarchy of which both the receiver and argument are some kind.

If the result is a kind of Set, then each element that occurs in both the receiver and *aBagOrSet* occurs exactly once in the result. If the result is a IdentityBag that is not an IdentitySet, and if an element occurs m times in the receiver and n times in the argument *aBagOrSet*, then the result contains the lesser of m or n occurrences of that element.

`+` *aBagOrSet* Union. Returns a kind of IdentityBag containing exactly the elements that are present in either the receiver or the argument *aBagOrSet*.

The class of the result is the lowest class in the hierarchy of which both the receiver and argument are some kind.

If the result is a kind of IdentitySet, then each element that occurs in either the receiver or *aBagOrSet* occurs exactly once in the result. If the result is an IdentityBag that is not an IdentitySet, and if an element occurs m times in the receiver and n times in the argument *aBagOrSet*, then the result contains m + n occurrences of that element.

`–` *aBagOrSet* Difference. Returns a kind of IdentityBag containing exactly those elements of the receiver that have a greater number of occurrences in the receiver than in the argument *aBagOrSet*. If an element occurs m times in the receiver and n times in *aBagOrSet* (where m >= n), then the result will contain m - n occurrences of that element.

The class of the result is the class of the receiver.

Updating

`at:` *anIndex* `put:` *anObject*

Disallowed. Generates an error, since the elements of an IdentityBag are not externally accessible through numeric indices.

Class Protocol

Backward Compatibility

Methods in this category are obsolete and are provided only for compatibility with earlier releases of GemStone. They will be removed in a future release.

`elementKind` Obsolete in GemStone 3.2.

Modifying Classes

`isModifiable` Returns true if the receiver and its array of named instance variables are modifiable.

IdentityCollisionBucket

An IdentityCollisionBucket is a CollisionBucket that is used in an IdentityKeyValueDictionary to store a collection of key/value pairs for which the keys hash to the same value. It provides support for the identity comparisons required by IdentityKeyValueDictionaries.

Superclasses CollisionBucket, AbstractCollisionBucket, Array, SequenceableCollection, Collection, Object

Named Instance Variables None

Instance Format Pointer, Indexable, Variant

Subclass Creation Allowed

Instance Protocol

Removing

`removeKey:` *aKey* `ifAbsent:` *aBlock*

Removes the key/value pair having the key *aKey*. If *aKey* is not found, returns the result of evaluting the zero-argument block *aBlock*.

Searching

`binarySearchForInsertKey:` *aKey*

Returns the negated index of *aKey* if found, or the offset at which to insert *aKey*.

`searchForKey:` *aKey* Returns the index of *aKey*, or nil if not found.

Class Protocol

Instance Creation

`new` Returns an IdentityCollisionBucket with a default capacity of four key/value pairs.

IdentityDictionary

IdentityDictionary is a Dictionary that is identity-based rather than equality-based.

IdentityDictionary implements key-value pairs by storing key-Association pairs. Each Association contains a key-value pair, and the key is duplicated in the dictionary or collision bucket for implementation reasons.

As with other identity-based collections, in an IdentityDictionary two keys or two values are considered to be the same only if they are identical; equivalent objects are not the same. Thus, if you add two key-value pairs to an IdentityDictionary and the keys are equivalent but not identical, then the result is that you have two pairs in the dictionary because the keys are not the same.

IdentityDictionary exhibits better performance than Dictionary and is to be preferred where it is appropriate.

Superclasses IdentityKeyValueDictionary, KeyValueDictionary, AbstractDictionary, Collection, Object

Named Instance Variables None

Instance Format Pointer, Indexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`associationAt:` *aKey*
Returns the Association with key *aKey*. Generates an error if no such Association exists.

`associationAt:` *aKey* `ifAbsent:` *aBlock*
Returns the Association with key *aKey*. If no such Association exists, returns the result of evaluating the zero-argument block *aBlock*.

`associationAt:` *aKey* `otherwise:` *defaultValue*
Returns the Association with key *aKey*. If no such Association exists, returns the given default value.

`at:` *aKey*
Returns the value of the Association with key *aKey*. Generates an error if no such Association exists.

`at:` *aKey* `ifAbsent:` *aBlock*

Returns the value that corresponds to *aKey*. If no such key/value pair exists, returns the result of evaluating the zero-argument block *aBlock*.

`at:` *aKey* `otherwise:` *aValue*

Returns the value that corresponds to *aKey*. If no such key/value pair exists, returns the given alternate value.

`keyAtValue:` *anObject* `ifAbsent:` *aBlock*

Returns the key of the first Association whose value matches the given object, *anObject*. If no match is found, this method evaluates the block *aBlock* and returns its result.

`keys` Returns an IdentitySet containing the receiver's keys.

`values` Returns an OrderedCollection containing the receiver's values.

Adding

`add:` *anAssociation* Requires an Association as the argument. If the receiver already includes an Association whose key is equal to that of *anAssociation*, this method redefines the value portion of that Association.

Enumerating

`associationsDo:` *aBlock*

Evaluates *aBlock* with each of the receiver's key/value pairs as the argument by creating an Association for each key/value pair. The argument *aBlock* must be a one-argument block. Returns the receiver.

`keysAndAssociationsDo:` *aBlock*

Evaluates *aBlock* with each of the receiver's key/Association pairs as the arguments. The argument *aBlock* must be a two-argument block. The first argument is the key and the second argument is the Association of each pair. Returns the receiver.

`keysAndValuesDo:` *aBlock*

Evaluates *aBlock* with each of the receiver's key/value pairs as the arguments. The argument *aBlock* must be a two-argument block. The first argument is the key and the second argument is the value of each key/value pair. Returns the receiver.

Hashing

`hashFunction:` *aKey*

The hash function should perform some operation on the value of the key (*aKey*) which returns a value in the range 1..tableSize.

`rebuildTable:` *newSize*

Rebuilds the hash table by saving the current state, initializing and changing the size of the table, and adding the key value pairs saved back to the dictionary.

Removing

`removeAssociation:` *anAssocation*

Removes an element from the receiver equal to anAssociation and returns anAssociation. If no such element is present, this method generates an error.

`removeAssociation:` *anAssocation* `ifAbsent:` *aBlock*

Removes *anAssocation* from the receiver. If no such element is present, evaluates the zero-argument block *aBlock* and returns the result of that evaluation.

`removeKey:` *aKey*

Removes the Association with key identical to *aKey* from the receiver and returns the value portion of that Association. If no Association is present with key identical to *aKey*, reports an error.

`removeKey:` *aKey* `ifAbsent:` *aBlock*

Removes the key/value pair with key *aKey* from the receiver and returns the value. If no key/value pair is present with key *aKey*, evaluates the zero-argument block *aBlock* and returns the result of that evaluation.

Searching

`collectAssociations:` *aBlock*

Evaluates *aBlock* with each of the receiver's associations as the argument. Collects the resulting values into a new dictionary and returns that dictionary. The argument *aBlock* must be a one-argument block.

`includesAssociation:` *anAssociation*

Returns true if the receiver contains an element identical to *anAssociation*. Returns false otherwise.

`includesKey:` *aKey* Reimplemented from KeyValueDictionary for efficiency.

Storing and Loading

`basicWriteTo:` *passiveObj*

Converts the receiver to its passive form and writes that information on *passiveObj*.

`loadVaryingFrom:` *passiveObj* `size:` *varyingSize*

Reads the varying part of the receiver from the given passive object. Does not record the receiver as having been read. Does not read the receiver's named instance variables, if any.

Updating

`addAssociation:` *anAssociation*

Adds the argument to the receiver.

`at:` *aKey* `put:` *aValue* If the receiver already contains a Association with the given key, this method makes *aValue* the value of that Association. Otherwise, it creates a new Association with the given key and value and adds it to the receiver. Returns *aValue*.

IdentityKeyValueDictionary

An IdentityKeyValueDictionary is a KeyValueDictionary that is an identity-based collection instead of equality-based. That is, two keys or two values are considered to be the same only if they are identical; equivalent objects are not the same. Thus, if you add two key-value pairs to an IdentityKeyValueDictionary and the keys are equivalent but not identical, then the result is that you have two pairs in the dictionary because the keys are not the same.

IdentityKeyValueDictionary exhibits better performance than KeyValueDictionary and is to be preferred where it is appropriate.

For multiuser applications that involve a lot of concurrent use of dictionaries, use RcKeyValueDictionary.

Superclasses KeyValueDictionary, AbstractDictionary, Collection, Object

Named Instance Variables None

Instance Format Pointer, Indexable, Variant

Subclass Creation Allowed

Instance Protocol

Comparing

`hash` Returns a numeric hash key for the receiver.

Hashing

`hashFunction:` *aKey* The hash function should perform some operation on the value of the key *aKey* which returns a value in the range 1..tableSize.

Initializing

`initialize:` *itsSize* Initializes the instance variables of the receiver to be an empty IdentityKeyValueDictionary of the specified size.

IdentitySet

An IdentitySet is an IdentityBag in which any distinct object can occur only once. Adding the same (identical) object to an IdentitySet multiple times is redundant. The result is the same as adding it once.

Since an IdentitySet is an identity-based collection, different (non-identical) but equivalent (equal) objects are treated as distinct from each other. In Sets, they are not distinct. Adding multiple equivalent objects to an IdentitySet yields an IdentitySet with as many elements as there are distinct equivalent objects. In short, two different elements of an IdentitySet are never identical, but they may be equivalent.

You can create subclasses of IdentitySet to restrict the kind of elements it contains. When creating a subclass of IdentitySet, you must specify a class as the aConstraint argument. This class is called the element kind of the new subclass. For each instance of the new subclass, the class of each element must be of the element kind.

Superclasses	IdentityBag, UnorderedCollection, Collection, Object
Named Instance Variables	None
Instance Format	Nsc, Nonindexable, Variant
Subclass Creation	Allowed

Instance Protocol

Adding

`add:` *anObject*
Adds *anObject* to the receiver only if it is not already an element of the receiver. Returns *anObject*. Has no effect if *anObject* is nil.

`add:` *anObject* `withOccurrences:` *anInteger*
Disallowed. Each element of an IdentitySet must be unique.

`addAll:` *aCollection*
Adds each element of *aCollection* to the receiver only if the element is not already present in the receiver. Occurrences of nil in *aCollection* are not added to the receiver.

`addValue:` *anObject*
Adds *anObject* to the receiver only if it is not already an element of the receiver, and if the receiver does not contain an equivalent object. Has no effect if *anObject* is nil. Returns *anObject*.

Integer

This is an abstract superclass that establishes protocol for all GemStone Smalltalk integers. Concrete subclasses include LargePositiveInteger, LargeNegativeInteger, and SmallInteger.

Superclasses Number, Magnitude, Object

Named Instance Variables None

Instance Format Pointer, Nonindexable, Variant

Subclass Creation Disallowed

Instance Protocol

Accessing

`denominator` For an Integer, always returns 1.

`numerator` For an Integer, always returns the receiver.

`size:` *anInteger* Disallowed. You may not change the size of an Integer.

Arithmetic

`*` *anInteger* Returns the product of the receiver and *anInteger*.

`+` *anInteger* Returns the sum of the receiver and *anInteger*.

`–` *anInteger* Returns the difference between the receiver and *anInteger*.

`/` *anInteger* Returns the result of dividing the receiver by *anInteger*.

`//` *anInteger* Divides the receiver by *anInteger*. Returns the integer quotient, with truncation toward negative infinity. For example,

```
9//4 = 2
-9//4 = -3
```

The selector `\\` returns the remainder from this division.

`factorial` Returns the factorial of the receiver. Returns 1 if the receiver is less than or equal to 1.

`quo:` *anInteger*
Divides the receiver by *anInteger*. Returns the integer quotient, with truncation toward zero. For example,

```
-9 quo: 4 = -2
```

The selector `rem:` returns the remainder from this division.

`\\` *anInteger*
Returns the modulus defined in terms of `//`. Returns a Number with the same sign as the argument *anInteger*. For example,

```
9\\4 = 1
-9\\4 = 3
9\\-4 = -3
```

Bit Manipulation

For purposes of bit manipulation, Integers are treated as two's-complement, infinite-precision binary numbers.

`allMask:` *anInteger*
Treats the argument *anInteger* as a bit mask. Returns true if all of the bits that are 1 in the argument are 1 in the receiver; returns false otherwise.

`anyMask:` *anInteger*
Treats the argument *anInteger* as a bit mask. Returns true if any of the bits that are 1 in the argument are 1 in the receiver; returns false otherwise.

`bitAnd:` *anInteger*
Returns an Integer whose bits are the logical and of the receiver's bits and the bits of *anInteger*.

`bitAt:` *i*
Returns the bit at the *i* th position of the receiver, where 0 is the least significant bit.

`bitInvert`
Returns an Integer whose bits are the one's-complement of the receiver.

`bitOr:` *anInteger*
Returns an Integer whose bits are the logical or of the receiver's bits and the bits of *anInteger*.

`bitShift:` *anInteger*
Returns an Integer whose value (in two's-complement representation) is the receiver's value (also in two's-complement representation) shifted by *anInteger* bits.

If *anInteger* is positive, the shift is to the left, and zero-bits enter at the right. If *anInteger* is negative, the shift is to the right, and the sign bit is repeated at the left.

`bitXor:` *anInteger*	Returns an Integer whose bits are the logical xor of the receiver's bits and the bits of *anInteger*.
`highBit`	(Subclass responsibility.) Returns the index of the high-order bit that is set in the binary representation of the receiver. (If the receiver is a negative integer, takes its absolute value first.) If the receiver is zero, this returns nil.
`noMask:` *anInteger*	Treats the argument *anInteger* as a bit mask. Returns true if none of the bits that are 1 in the argument are 1 in the receiver; returns false otherwise.

Comparing

`<` *anInteger*	Returns true if the receiver is less than *anInteger*; returns false otherwise.
`<=` *anInteger*	Returns true if the receiver is less than or equal to *anInteger*; returns false otherwise.
`=` *anInteger*	Returns true if the receiver is equal to *anInteger*; returns false otherwise.
`>` *anInteger*	Returns true if the receiver is greater than *anInteger*; returns false otherwise.
`>=` *anInteger*	Returns true if the receive is greater than *anInteger*; returns false otherwise.
`~=` *anInteger*	Returns true if the receiver is not equal to *anInteger*; returns false otherwise.

Converting

`asCharacter`	Returns the Character whose value equals the receiver. Allowable range for the receiver is 0 to 65535, inclusive.
`asDecimalFloat`	Returns a DecimalFloat representing the receiver.
`asFloat`	Returns a Float representing the receiver.
`asFraction`	Returns a Fraction having a numerator equal to the receiver and a denominator of 1.
`asHexString`	Returns a String representing the receiver as a base-16 number.
`asInteger`	Returns the receiver.

`asJISCharacter` Returns the JISCharacter whose JIS value equals the receiver. Allowable range for the receiver is 0 to 65535, inclusive.

Divisibility

`gcd:` *anInteger* Returns the greatest common divisor of the receiver and *anInteger*.

`lcm:` *anInteger* Returns the least common multiple of the receiver and *anInteger*.

Flow of Control

`to:` *aNumber* Returns an Array containing all Integers between the receiver and the argument.

`to:` *aNumber* `by:` *interval*
Returns an Array containing all Integers between the receiver and the argument, skipping the given *interval*.

Formatting

`asString` Returns a string representing the receiver. Positive values do not include a leading + .

`printOn:` *aStream* `base:` *b*
Prints a representation of the receiver on *aStream* in base *b*. The base b must be 2 <= *b* <= 36. Returns the receiver.

`printOn:` *aStream* `base:` *b* `showRadix:` *aBoolean*
Prints a representation of the receiver on *aStream* in base *b*. The base b must be 2 <= *b* <= 36. Returns the receiver.

`printStringRadix:` *base*
Returns a String that describes the receiver in the specified radix.

`printStringRadix:` *base* `showRadix:` *aBoolean*
Returns a String that describes the receiver in the specified radix.

Truncation and Rounding

`ceiling`	Returns the receiver.
`floor`	Returns the receiver.
`rounded`	Returns the receiver.
`truncated`	Returns the receiver.

Class Protocol

Instance Creation

`fromCompleteString:` *aString*

Returns an instance of the appropriate subclass, reconstructed from *aString*. Leading blanks and trailing blanks are permitted. Trailing non-digits generate an error.

Smalltalk radix syntax for non-base-10 numbers is supported.

`fromHexString:` *aString* Returns an instance of the appropriate subclass of the receiver whose value is read from the given string.

`fromStream:` *aStream* Reads bytes from *aStream* and returns an instance of the appropriate subclass. Starting at the current position of *aStream*, leading blanks and trailing blanks are permitted. Trailing non-digits terminate the conversion without raising any errors.

Smalltalk radix syntax for non-base-10 numbers is supported.

`fromString:` *aString* Returns an instance of the appropriate subclass, reconstructed from *aString*. Leading blanks and trailing blanks are permitted. Trailing non-digits terminate the conversion without raising any errors.

Smalltalk radix syntax for non-base-10 numbers is supported.

IntegerKeyValueDictionary

An IntegerKeyValueDictionary is a KeyValueDictionary in which the keys are Integers.

One useful application of IntegerKeyValueDictionary is as an implementation of a sparse array. If the keys to the dictionary are normal array indexes and the array indexes used are sparsely scattered over the range of the array, then IntegerKeyValueDictionary can provide a fast implementation that has much lower storage costs.

Superclasses KeyValueDictionary, AbstractDictionary, Collection, Object

Named Instance Variables None

Instance Format Pointer, Indexable, Variant

Subclass Creation Allowed

Instance Protocol

Updating

`at:` *aKey* `put:` *aValue* Stores the aKey/aValue pair in the KeyValueDictionary. Generates an error if *aKey* is not a kind of Integer.

Interval

Superclasses SequenceableCollection, Collection, Object

Named Instance Variables

from — Initial number in the sequence.

to — Last number in the sequence.

by — Increment for determining the next number in the sequence.

Instance Format Pointer, Indexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`at:` *anIndex* — Intervals cannot be accessed by integer offset.

`increment` — Returns a Number which represents the step size in the arithmetic progression represented by the receiver.

`size` — Returns a count of the number of elements in the arithmetic progression represented by the receiver.

Comparing

`=` *anInterval* — Returns true if the receiver is equal to the argument, false otherwise.

`hash` — Returns some Integer related to the contents of the receiver. If two objects compare equal (=) to each other, the results of sending hash to each of those objects must also be equal.

Concatenating

`,` *aCollection* — Returns an Array that contains the elements of the receiver followed by the elements of *aCollection*. The receiver's standard enumeration order is used.

Notice that the result of this method is an Array, whereas the addAll: method modifies the receiver, an Interval.

Copying

`copyFrom:` *startIndex* `to:` *stopIndex*
Returns an Array containing the elements of the receiver between start and stop.

`copyReplaceAll:` *oldSubCollection* `with:` *newSubCollection*
Returns an Array in which all sequences of *oldSubCollection* contained within the receiver have been replaced by elements of *newSubCollection*.

`copyReplaceFrom:` *startIndex* `to:` *stopIndex* `with:` *replacementElements*
Returns an Array in which all elements in the receiver between indexes *startIndex* and *stopIndex* inclusive have been replaced by those contained in aSequenceableCollection.

`copyReplaceFrom:` *startIndex* `to:` *stopIndex* `withObject:` *anObject*
Returns an Array in which all elements in the receiver between indexes *startIndex* and *stopIndex* inclusive have been replaced by *anObject*.

`copyWith:` *anObject* Returns an Array containing the elements of the receiver with *anObject* appended at the end.

`copyWithout:` *anObject* Returns an Array that contains all the elements of the receiver except *anObject*.

`reverse` Returns an Array containing containing the elements of the receiver in the reverse order.

Eumerating

`collect:` *aBlock* Evaluates *aBlock* with each of the receiver's elements as the argument. Collects the resulting values into an Array and returns the Array. The argument *aBlock* must be a one-argument block.

The result preserves the ordering of the receiver. That is, if element a comes before element b in the receiver, then element a is guaranteed to come before b in the result.

`do:` *aBlock* Evaluates the one-argument block *aBlock* using each element of the receiver in order. Returns the receiver.

`reject:` *aBlock*
Evaluates *aBlock* with each of the receiver's elements as the argument. Stores the values for which *aBlock* is false into an Array and returns the Array. The argument *aBlock* must be a one-argument block.

The result preserves the ordering of the receiver. That is, if element a comes before element b in the receiver, then element a is guaranteed to come before b in the result.

`select:` *aBlock*
Evaluates *aBlock* with each of the receiver's elements as the argument. Stores the values for which *aBlock* is true into an Array and returns the Array. The argument *aBlock* must be a one-argument block.

The result preserves the ordering of the receiver. That is, if element a comes before element b in the receiver, then element a is guaranteed to come before b in the result.

Updating

`at:` *anIndex* `put:` *aValue*
Intervals cannot be accessed by integer offset.

Class Protocol

Instance Creation

`from:` *start* `to:` *stop*
Returns an Interval that represents an arithmetic progression from *start* to *stop* in increments of one.

`from:` *start* `to:` *stop* `by:` *step*
Returns an Interval that represents an arithmetic progression from *start* to *stop* in increments of *step*.

`new`
Intervals cannot be created by the method `new`.

`new:` *anInteger*
Intervals cannot be created by the method `new:`.

InvariantArray

An InvariantArray is an Array that is made to be invariant (immutable) when it is committed, if it is not already invariant before then. Unless otherwise handled explicitly, an InvariantArray that has not yet been committed is still modifiable. Array literals are always invariant; if you assign one to a variable, then the object to which the variable refers cannot be changed.

Immutability implies that the Array's size, ordering of elements, and element membership cannot be changed. Thus, if an object is an element of the Array, that same object must remain an element of the Array, and it must remain located at its current index. Immutability does not imply that the element object itself cannot be changed, but only that its relation to the Array is immutable.

Superclasses	Array, SequenceableCollection, Collection, Object
Named Instance Variables	None
Instance Format	Pointer, Indexable, Invariant
Subclass Creation	Allowed

InvariantEUCString

This class represents an invariant Japanese string in Extended Unix Code format.

Superclasses EUCString, JapaneseString, CharacterCollection, SequenceableCollection, Collection, Object

Named Instance Variables None

Instance Format Byte, Indexable, Invariant

Subclass Creation Allowed

Instance Protocol

Formatting

`asEUCString` Returns an EUCString representing the receiver.

InvariantString

InvariantString is a subclass of String for which all instances are immutable after they are committed.

Superclasses String, CharacterCollection, SequenceableCollection, Collection, Object

Named Instance Variables None

Instance Format Byte, Indexable, Invariant

Subclass Creation Allowed

Instance Protocol

Formatting

`asString` Returns a copy of the receiver as an instance of class String.

ISOLatin

ISOLatin is a subclass of String which provides means for GemStone Smalltalk applications to extend the behavior of String with native-language-specific sorting or other behavior.

ISOLatin inherits all of its behavior from String, and thus inherits the the English ASCII defaults for character set interpretation, as provided by the C runtime library and by Unix.

It is the user's responsibility to create a subclass of ISOLatin and implement appropriate comparison methods in the subclass if an application requires language-specific sorting or comparison semantics.

Superclasses	String, CharacterCollection, SequenceableCollection, Collection, Object
Named Instance Variables	None
Instance Format	Byte, Indexable, Variant
Subclass Creation	Allowed

Instance Protocol

Formatting

`printOn:` *aStream*	Puts a displayable representation of the receiver on the given stream.

JapaneseString

This class represents behavior common to all JapaneseString classes.

Subclasses must reimplement the following selectors:

```
size
size:
at:
at:put:
```

However these selectors do not generate the subclass responsibility error (error 2008) because to do so would break `Object | printString` method.

Superclasses CharacterCollection, SequenceableCollection, Collection, Object

Named Instance Variables None

Instance Format Byte, Indexable, Variant

Subclass Creation Allowed

Instance Protocol

Adding

`insertAll:` *aCharOrCharColl* `at:` *anIndex*

Inserts *aCharOrCharColl* at the specified index. Returns *aCharOrCharColl*.

Concatenating

`,` *aCharOrCharCollection*

Returns a new instance of the receiver's class that contains the elements of the receiver followed by the elements of *aCharOrCharCollection*.

Warning: Creating a new instance and copying the receiver take time. If you can safely modify the receiver, it can be much faster to use the `addAll:` method. See the documentation of the Concatenating category of class SequenceableCollection for more details.

Converting

`asEUCString`

Returns an EUCString representing the receiver.

Formatting

`printOn:` *aStream*	Puts a displayable representation of the receiver on the given stream.
`printString`	Returns a String whose contents are a displayable representation of the receiver.

Searching

`findString:` *subString* `startingAt:` *startIndex*

If the receiver contains *subString* beginning at some point at or after *startIndex*, returns the index at which *subString* begins. If the receiver does not contain *subString*, returns zero.

Storing and Loading

`loadFrom:` *passiveObj*	Reads from *passiveObj* the passive form of an object. Converts the object to its active form by loading the information into the receiver.
`writeTo:` *passiveObj*	Converts the receiver to its passive form and writes that information on *passiveObj*.

JISCharacter

The Japanese Industrial Standards organization (JIS) has defined a standard Japanese character set, containing codes for thousands of characters. Some characters have both a one-byte and a two-byte representation. These characters include Roman characters, digits, the space character, katakana, and some punctuation and special characters. Both the one-byte and two-byte representations of the same character can be freely mixed in Japanese text.

There are 65535 instances of JISCharacter. You may not create new instances of JISCharacter. All instances of a given JIS character are both equal (=) and identical (==).

Superclasses AbstractCharacter, Magnitude, Object

Named Instance Variables None

Instance Format Special, Nonindexable, Invariant

Subclass Creation Disallowed

Instance Protocol

Accessing

`eucValue` Returns the EUC code of the receiver as a SmallInteger.

`firstByte` Returns a SmallInteger representing the value of the leading byte of the JIS value of the receiver. If the receiver is a one byte JISCharacter, returns zero.

`jisValue` Returns the JIS code of the receiver as a SmallInteger.

`secondByte` Returns a SmallInteger representing the value of the last byte of the JIS value of the receiver.

Comparing

`< aCharacter` Returns true if the receiver is less than the argument. Returns false otherwise.

`<= aCharacter` Returns true if the receiver is less than or equal to the argument. Returns false otherwise.

`= aCharacter` Returns true if the receiver is equal to the argument. Returns false otherwise.

`>` *aCharacter*	Returns true if the receiver is greater than the argument. Returns false otherwise.
`>=` *aCharacter*	Returns true if the receiver is greater than or equal to the argument. Returns false otherwise.

Converting

`asCharacter`	Returns the (ASCII) Character corresponding to the receiver.
`asciiValue`	Returns the ASCII value (a SmallInteger) corresponding to the receiver.
`asHankaku`	Returns a JISCharacter that is the one byte representation of a two byte digit or two byte Roman receiver. If the receiver is not a two byte digit or a two byte Roman character, returns the receiver.
`asInteger`	Returns the JIS code of the receiver as a SmallInteger.
`asJapaneseString:` *aClass*	Returns an instance of class *aClass* containing only the receiver. *aClass* must be a subclass of JapaneseString.
`asJISCharacter`	Returns the receiver.
`asLowercase`	Returns a JISCharacter that is the lowercase character corresponding to the receiver. If the receiver is lowercase or has no case, this returns the receiver itself. Returns a one-byte lowercase Roman character if the receiver is a one byte uppercase Roman character. Returns a two-byte lowercase Roman character if the receiver is a two byte uppercase Roman character. The only JISCharacters that have case distinction are Roman, Greek, and Russian.
`asUppercase`	Returns a JISCharacter that is the uppercase character corresponding to the receiver. If the receiver is uppercase or has no case, this returns the receiver itself. Returns a one-byte uppercase Roman character if the receiver is a one byte lowercase Roman character. Returns a two-byte uppercase Roman character if the receiver is a two-byte lowercase Roman character. The only JISCharacters that have case distinction are Roman, Greek, and Russian.

`asZenkaku`	Returns a JISCharacter that is the two byte representation of a one byte digit or one byte Roman receiver. If the receiver is not a one byte digit or one byte Roman character, returns the receiver.
`digitValue`	Returns a SmallInteger representing the value of the receiver, a digit, or returns nil if the receiver is not a digit.

Copying

`copy`	Returns the receiver. (Does not create a new instance.)

Formatting

`asEUCString`	Returns an EUCString that represents the receiver.
`asString`	Returns a one-character instance of String containing the receiver.
`displayWidth`	Returns the width necessary to display the receiver. For a JISCharacter, this can be 1 or 2.
`printOn:` *aStream*	Puts a displayable representation of the receiver on the given stream.

Storing and Loading

`writeTo:` *passiveObj*	Converts the receiver to its passive form and writes that information on *passiveObj*.

Testing

`hasEUCFormat`	Returns true if the receiver can be represented in EUC format. Returns false otherwise.
`isDigit`	Returns true if the receiver is a one or two byte digit. Returns false otherwise.
`isEquivalent:` *aCharacter*	Returns true if the receiver is the same character as the argument regardless of case or internal representation.
`isFirstLevelKanji`	Returns true if the receiver is a first level kanji character. Returns false otherwise.
`isGreek`	Returns true if the receiver is a Greek character. Returns false otherwise.

`isHiragana`	Returns true if the receiver is a hiragana character. Returns false otherwise.
`isJisAscii`	Returns true if the receiver is a JIS-ASCII character. Returns false otherwise.
`isKana`	Returns true if the receiver is a hiragana character or a one or two byte katakana character. Returns false otherwise.
`isKanji`	Returns true if the receiver is a kanji character. Returns false otherwise.
`isKatakana`	Returns true if the receiver is a one or two byte katakana character. Returns false otherwise.
`isLineElement`	Returns true if the receiver is a line element character. Returns false otherwise.
`isLowercase`	Returns true if the receiver is a lowercase character. Returns false otherwise. The only JISCharacters that have case distinction are Roman, Greek and Russian.
`isLowercaseGreek`	Returns true if the receiver is a lowercase Greek character. Returns false otherwise.
`isLowercaseRussian`	Returns true if the receiver is a lowercase Russian character. Returns false otherwise.
`isOneByteCharacter`	Returns true if the receiver is a one byte character. Returns false otherwise.
`isOneByteDigit`	Returns true if the receiver is a one byte digit. Returns false otherwise.
`isOneByteKatakana`	Returns true if the receiver is a one byte katakana character. Returns false otherwise.
`isOneByteLowercaseRoman`	Returns true if the receiver is a one byte lowercase Roman character. Returns false otherwise.
`isOneByteRoman`	Returns true if the receiver is a one byte Roman character. Returns false otherwise.
`isOneByteUppercaseRoman`	Returns true if the receiver is a one byte uppercase Roman character. Returns false otherwise.

`isRussian`	Returns true if the receiver is a Russian character. Returns false otherwise.
`isSecondLevelKanji`	Returns true if the receiver is a second level kanji character. Returns false otherwise.
`isSpecialChar`	Returns true if the receiver is a special character. Returns false otherwise.
`isTwoByteCharacter`	Returns true if the receiver is a two byte character. Returns false otherwise.
`isTwoByteDigit`	Returns true if the receiver is a two byte digit. Returns false otherwise.
`isTwoByteKatakana`	Returns true if the receiver is a two byte katakana character. Returns false otherwise.
`isTwoByteLowercaseRoman`	Returns true if the receiver is a two byte lowercase Roman character. Returns false otherwise.
`isTwoByteRoman`	Returns true if the receiver is a two byte Roman character. Returns false otherwise.
`isTwoByteUppercaseRoman`	Returns true if the receiver is a two byte uppercase Roman character. Returns false otherwise.
`isUppercase`	Returns true if the receiver is a uppercase character. Returns false otherwise. The only JISCharacters that have case distinction are Roman, Greek and Russian.
`isUppercaseGreek`	Returns true if the receiver is an uppercase Greek character. Returns false otherwise.
`isUppercaseRussian`	Returns true if the receiver is an uppercase Russian character. Returns false otherwise.

Class Protocol

Instance Creation

`fromStream:` *aStream* Returns the next JISCharacter in the stream *aStream*.

`fromString:` *aJapaneseString*
If *aJapaneseString* is a one-character JapaneseString, returns the character in *aJapaneseString*. Otherwise, generates an error.

`new` Disallowed. You may not create new instances of JISCharacter.

`withEUCValue:` *anEUCValue*
Returns the JISCharacter with the specified EUC value.

`withValue:` *ajisValue* Returns the JISCharacter with the specified JIS value. Generates an error if *ajisValue* is greater than 65535 or less than 0.

Non-Printable Characters

`backspace` Returns the JIS backspace character.

`cr` Returns the JIS carriage return character.

`esc` Returns the JIS escape character.

`lf` Returns the JIS linefeed character.

`newPage` Returns the JIS new-page character.

`space` Returns the JIS one byte space character.

`tab` Returns the JIS tab character.

Printable Characters

`lowercaseGreek` Returns an Array containing all lowercase Greek JIS characters in alphabetic order.

`lowercaseRussian` Returns an Array containing all lowercase Russian JIS characters in alphabetic order.

`oneByteDigits` Returns an Array containing one byte JISCharacters representing the digits 0 through 9.

`oneByteLowercaseRoman`
Returns an Array containing all one byte lowercase Roman JISCharacters in alphabetic order.

`oneByteUppercaseRoman`
Returns an Array containing all one byte uppercase Roman JISCharacters in alphabetic order.

`twoByteDigits` Returns an Array containing two byte JISCharacters representing the digits 0 through 9.

`twoByteLowercaseRoman`
Returns an Array containing all two byte lowercase Roman JISCharacters in alphabetic order.

`twoByteUppercaseRoman`
Returns an Array containing all two byte uppercase Roman JISCharacters in alphabetic order.

`uppercaseGreek` Returns an Array containing all uppercase Greek JIS characters in alphabetic order.

`uppercaseRussian` Returns an Array containing all uppercase Russian JIS characters in alphabetic order.

Storing and Loading

`loadFrom:` *passiveObj* Reads from *passiveObj* the passive form of an object. Converts the object to its active form by loading the information into a new instance of the receiver. Returns the new instance.

JISString

JISString represents Japanese strings containing JISCharacters. Each character in a JISString occupies 2 bytes.

Superclasses JapaneseString, CharacterCollection, SequenceableCollection, Collection, Object

Named Instance Variables None

Instance Format Byte, Indexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`at:` *anIndex* Returns the JISCharacter at *anIndex*.

`size` Returns the size of the receiver in characters.

Adding

`add:` *aCharOrCharCollection*

Appends *aCharOrCharCollection* to the receiver. The argument *aCharOrCharCollection* must be a CharacterCollection or a kind of AbstractCharacter.

`addAll:` *aCharOrCharCollection*

Equivalent to `add:` *aCharOrCharCollection*.

`addLast:` *aCharOrCharCollection*

Equivalent to `add:` *aCharOrCharCollection*.

`insertAll:` *aCharOrCharCollection* `at:` *anIndex*

Inserts *aCharOrCharCollection* at the specified index.

Concatenating

`,` *aCharOrCharColl* Returns a new instance of the receiver's class that contains the elements of the receiver followed by the elements of *aCharOrCharColl*.

Note: Creating a new instance and copying the receiver take time. If you can safely modify the receiver, it is faster to use the `addAll:` method.

Converting

`asSymbol` Returns the receiver as a DoubleByteSymbol.

`asSymbolKind` Returns a copy of the receiver as an instance of class Symbol.

Formatting

`asJISString` Returns the receiver.

`printString` Returns a JISString whose contents are a displayable representation of the receiver.

Searching

`indexOf:` *aCharacter* `startingAt:` *startIndex*

Returns the index of the first occurrence of *aCharacter* in the receiver, not preceding *startIndex*. If the receiver does not contain *aCharacter*, this returns zero.

Updating

`at:` *anIndex* `put:` *aCharacter*

Stores *aCharacter* at the specified location.

`size:` *anInteger* Changes the size of the receiver to *anInteger*.

If *anInteger* is less than the current size of the receiver, the receiver is shrunk accordingly. If *anInteger* is greater than the current size of the receiver, the receiver is extended and new elements are initialized to nil.

KeyValueDictionary

KeyValueDictionary is a concrete subclass of AbstractDictionary. In a KeyValueDictionary, keys may be of mixed classes.

A KeyValueDictionary stores key-value pairs under an index that is generated by applying a hash function to the key; it does not use Associations. The hashing improves retrieval speed. However, you must observe an important restriction: after a key/value pair has been added to a KeyValueDictionary, you must not modify the key. Doing so renders the value inaccessible.

A KeyValueDictionary is also an equality-based collection. That is, two keys or two values are considered to be the same if they are equivalent; they need not be identical to be the same. Thus, if you add two key-value pairs to a KeyValueDictionary but the keys are equivalent, even if they are not identical, then the result is that the second pair overwrites the first one, because the keys are the same.

Some other kinds of dictionaries are identity-based rather than equality-based. These other kinds of dictionaries exhibit better performance than KeyValueDictionary and are to be preferred where they are appropriate.

For multiuser applications that involve a lot of concurrent use of dictionaries, use RcKeyValueDictionary.

Superclasses AbstractDictionary, Collection, Object

Named Instance Variables

numElements — A SmallInteger that represents the number of key/value pairs in the dictionary.

numCollisions — A SmallInteger that represents the cumulative number of collisions that have occurred during the addition of the elements to the dictionary since the last rebuild.

collisionLimit — A SmallInteger that represents the number of collisions allowed before rebuilding the hash table.

tableSize — A SmallInteger that represents the size of the primary hash table.

Instance Format Pointer, Indexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`at:` *aKey* `ifAbsent:` *aBlock*

Returns the value that corresponds to *aKey*. If no such key/value pair exists, returns the result of evaluating the zero-argument block *aBlock*.

`at:` *aKey* `otherwise:` *aValue*

Returns the value that corresponds to *aKey*. If no such key/value pair exists, returns the given alternate value.

`collisionLimit` Returns the value of the **collisionLimit** instance variable.

`keyAt:` *aKey* `otherwise:` *aValue*

Returns the key that corresponds to *aKey*. If no such key/value pair exists, returns the given alternate value.

Note that the method `keyAt:` is private and does not have behavior that is compatible with this method.

`keys` Returns an IdentitySet containing the receiver's keys.

`numCollisions` Returns the total number of collisions, the sum of the number of key/value pairs in all collision buckets.

`numElements` Same as the size method.

`tableSize` Returns the size of the primary hash table.

Clustering

`clusterDepthFirst` This method clusters the receiver and its named instance variables and the key/value pairs in depth-first order. Returns true if the receiver has already been clustered during the current transaction; returns false otherwise.

Copying

`copy` Returns a copy of the receiver.

Enumerating

`associationsDo:` *aBlock*

Evaluates *aBlock* with each of the receiver's key/value pairs as the argument by creating an Association for each key/value pair. The argument *aBlock* must be a one-argument block.

`keysAndValuesDo:` *aBlock*

Evaluates *aBlock* with each of the receiver's key/value pairs as the arguments. The argument *aBlock* must be a two-argument block. The first argument is the key and the second argument is the value of each key/value pair.

`keysDo:` *aBlock* — For each key/value pair in the receiver, evaluates the one-argument block *aBlock* with the key as the argument.

`valuesDo:` *aBlock* — For each key/value pair in the receiver, evaluates the one-argument block *aBlock* with the value as the argument.

Hashing

`rebuildTable:` *newSize* — Rebuilds the hash table by saving the current state, initializing and changing the size of the table, and adding the key value pairs saved back to the hash dictionary.

Initializing

`initialize:` *itsSize* — Initializes the instance variables of the receiver to be an empty KeyValueDictionary of the specified size.

`tableSize:` *newSize* — Sets the table size to a new value.

Removing

`removeKey:` *aKey* `ifAbsent:` *aBlock*

Removes the key/value pair with key *aKey* from the receiver and returns the value. If no key/value pair is present with key *aKey*, evaluates the zero-argument block *aBlock* and returns the result of that evaluation.

Searching

`collect:` *aBlock* — Evaluates *aBlock* with each of the receiver's values as the argument. Collects the resulting values into a new hash dictionary that this method returns. The argument *aBlock* must be a one-argument block.

Statistics

`statistics` Returns a Dictionary containing statistics that can be useful in determining the performance of a hash dictionary. The dictionary contains the following information:

TotalCollisionBuckets - The number of collision buckets required to implement the hash dictionary.

AveragePairsPerBucket - The average number of key/value pairs in each bucket.

LargestBucket - The bucket having the most key/value pairs. This bucket contains the most keys for which the hash function did not provide a good distribution over the range of values in the table.

Storing and Loading

`basicWriteTo:` *passiveObj*

Converts the receiver to its passive form and writes that information on *passiveObj*.

`loadNamedIVsFrom:` *passiveObj*

Reads named instance variables from the given passive object. The first instance variable should already have been parsed and be available in the *passiveObj* argument.

`loadVaryingFrom:` *passiveObj* `size:` *varyingSize*

Reads the varying part of the receiver from the given passive object. Does not record the receiver as having been read. Does not read the receiver's named instvars, if any.

Updating

`at:` *aKey* `put:` *aValue* Stores the aKey/aValue pair in the hash dictionary. Rebuilds the hash table if the addition caused the number of collisions to exceed the limit allowed.

`changeToSegment:` *segment*

Assigns the receiver and any collision buckets to the given *segment*.

`collisionLimit:` *aLimit*

Sets the collision limit to *aLimit*.

Class Protocol

Accessing the Class Format

`firstPublicInstVar` Returns the index of the first user-visible instance variable defined in this class, whether or not this class actually has user-visible instance variables.

`hasPublicInstVars` Returns true if this class has user-visible instance variables defined, false if not.

Instance Creation

`new` Creates an instance of KeyValueDictionary with a default table size.

`new:` *tableSize* Creates an instance of KeyValueDictionary based upon the specified table size.

Storing and Loading

`loadFrom:` *passiveObj* `mappingToClass:` *newClass*

Reads from *passiveObj* the passive form of an object with named instance variable format. Converts the object to its active form by loading the information into a new instance of the receiver. Returns the new instance.

LanguageDictionary

A LanguageDictionary is a SymbolDictionary designed to hold language-dependent objects. Language symbols are used as keys, so that a user's language symbol (see `UserProfile | nativeLanguage`) can be used to find the appropriate object for the native language.

Superclasses SymbolDictionary, IdentityDictionary, IdentityKeyValueDictionary, KeyValueDictionary, AbstractDictionary, Collection, Object

Named Instance Variables None

Instance Format Pointer, Indexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`value` Returns the value associated with the user's native language Symbol. That Symbol is taken from the user's UserProfile object. If there is no entry for the given language, an error is generated.

LargeNegativeInteger

Instances of LargeNegativeInteger represent negative integers whose values are less than the minimum SmallInteger (-(2 to the 29th power)). Each instance of LargeNegativeInteger is stored as an Array of bytes, where each double-byte pair represents a base 32768 digit. The first two bytes in that Array constitute the least significant base 32768 digit, and the last two bytes are the most significant base 32768 digit. Within each digit, the least significant byte is first, followed by the more significant byte.

Coercion between LargeNegativeInteger and SmallInteger occurs automatically.

Superclasses Integer, Number, Magnitude, Object

Named Instance Variables None

Instance Format Byte, Indexable, Variant

Subclass Creation Disallowed

Instance Protocol

Accessing

`at:` *anIndex* Disallowed.

`at:` *anIndex* `put:` *aNumber*
Disallowed. You may not change the value of a LargeNegativeInteger.

Arithmetic

`abs` Returns a LargePositiveInteger that is the absolute value of the receiver.

`negated` Returns LargePositiveInteger that is the negation of the receiver.

Bit Manipulation

`highBit` Returns the index of the high-order bit that is set in the binary representation of the receiver. (Because the receiver is a negative integer, this method takes its absolute value first.)

Testing

`negative`	Returns true.
`positive`	Returns false.

Truncation and Rounding

`truncated`	Returns the receiver.

LargePositiveInteger

Instances of LargePositiveInteger represent positive integers whose values are greater than the maximum SmallInteger (2 to the 29th power minus 1). Each instance of LargePositiveInteger is stored as an Array of bytes, where each double-byte pair represents a base 32768 digit. The first two bytes in that Array constitute the least significant base 32768 digit, and the last two bytes are the most significant base 32768 digit. Within each digit, the least significant byte is first, followed by the more significant byte.

Coercion between LargePositiveInteger and SmallInteger occurs automatically.

Superclasses Integer, Number, Magnitude, Object

Named Instance Variables None

Instance Format Byte, Indexable, Variant

Subclass Creation Disallowed

Instance Protocol

Accessing

`at:` *anIndex* Disallowed.

`at:` *anIndex* `put:` *aNumber*
Disallowed. You may not change the value of a LargePositiveInteger.

Arithmetic

`abs` Returns a LargePositiveInteger that is the absolute value of the receiver.

`negated` Returns a LargeNegativeInteger that is the negation of the receiver.

Bit Manipulation

`highBit` Returns the index of the high-order bit that is set in the binary representation of the receiver.

Testing

`negative`	Returns false.
`positive`	Returns true.

Truncation and Rounding

`truncated`	Returns the receiver.

Magnitude

Magnitude is an abstract superclass that defines methods for kinds of objects that are ordered. Concrete subclasses of Magnitude include Character, DateTime, and Integer.

Superclasses	Object
Named Instance Variables	None
Instance Format	Pointer, Nonindexable, Variant
Subclass Creation	Allowed

Instance Protocol

Comparing

A subclass of Magnitude must implement two comparison operators: equality (=) and less-than (<). The remaining comparison operators are defined in terms of these two operations, so the subclass can inherit their definitions. Each of these methods generates an error if the argument aMagnitude cannot be compared with the receiver.

`<` *aMagnitude*	(Subclass responsibility.) Returns true if the receiver is less than *aMagnitude*; returns false otherwise.
`<=` *aMagnitude*	Returns true if the receiver is less than or equal to *aMagnitude*; returns false otherwise.
`=` *aMagnitude*	(Subclass responsibility.) Returns true if the receiver is equal to *aMagnitude*; returns false otherwise.
`>` *aMagnitude*	Returns true if the receiver is greater than *aMagnitude*; returns false otherwise.
`>=` *aMagnitude*	Returns true if the receiver is greater than or equal to *aMagnitude*; returns false otherwise.
`between:` *min* `and:` *max*	Returns true if the receiver is less than or equal to the argument *max* and greater than or equal to the argument *min*. Returns false otherwise.
`hash`	(Subclass responsibility.) Returns a numeric hash index.

`max:` *aMagnitude*	Returns the receiver or the argument, whichever has the greater magnitude.
`min:` *aMagnitude*	Returns the receiver or the argument, whichever has the lesser magnitude.

Class Protocol

Instance Creation

`fromStream:` *aStream*	(Subclass responsibility.) Returns an instance of the receiver.
`fromString:` *aString*	Returns an instance of the appropriate subclass, reconstructed from *aString*. The String must contain only characters representing the object to be created, although leading and trailing blanks are permitted.

Metaclass

Each Metaclass describes the protocol of its single instance, which is a Class. The class methods of GemStone kernel classes are actually stored as instance methods of those classes' Metaclasses.

For example, 3 is an instance of the class SmallInteger. SmallInteger is an instance of the Metaclass SmallInteger class, and describes the protocol of all SmallIntegers. SmallInteger class is itself an instance of Metaclass, and describes the protocol (that is, the class methods) of the class SmallInteger.

Consider the following example. The description of class SmallInteger contains two kinds of protocol: instance methods and class methods. Instance methods are understood by SmallIntegers (instances of class SmallInteger). Class methods are understood by the class-defining object SmallInteger itself (which is the single instance of the Metaclass SmallInteger class, and inherits its protocol from Class, Behavior, and Object).

Superclasses Behavior, Object

Named Instance Variables **thisClass —** The Class that this instance of Metaclass describes.

Instance Format Pointer, Nonindexable, Variant

Subclass Creation Disallowed

Instance Protocol

Accessing

`classHistory` Returns the **classHistory** for the Class of which the receiver is a Metaclass.

`extraDict` Returns the **extraDict** of the receiver's sole instance. See `Class | extraDict`.

`name` Returns the name of the receiver. For example, SmallInteger class name returns SmallInteger class (the receiver, a Metaclass).

`thisClass` Returns the Class of which the receiver is a Metaclass. For example, `SmallInteger class thisClass` returns SmallInteger (the Class).

Class Instance Variables

`addInstVarNames:` *instVarNamesArray*

Adds the instance variables specified in the argument to the receiver and any of its subclasses. Generates an error upon encountering a name that is not a valid instance variable name or that is already an instance variable of the receiver.

Instance variables that are added to a Metaclass are called Class Instance Variables.

Clustering

`clusterDepthFirst`

Overrides the inherited method. This method clusters, in depth-first order, the receiver's classVars, methodDict, and categories instance variables. (This method does not cluster the instance variables superclass, which may be shared with other Metaclasses; instVarNames and constraints, which are shared among all Metaclasses; and poolDictionaries, which are shared among an arbitrary number of Behaviors.) The receiver itself is not clustered.

Returns true if the receiver has already been clustered during the current transaction; returns false otherwise.

`clusterDescription`

Overrides the inherited method. For instances of Metaclass, only the classVars and categories instance variables are clustered. (The instance variables instVarNames and constraints are not clustered since they are shared among all Metaclasses.)

Displaying

`instanceString`

Returns a string that can be used to name an instance of the receiver. Since the receiver has one instance, returns the name of that instance.

`instanceSymbol`

Returns a symbol that can be used to name an instance of the receiver. Since a Metaclass has only one instance, returns the name of that instance.

Instance Creation

`new` — Disallowed. To create a new Class or Metaclass, use `Class | subclass:instVarNames:..` instead.

`new:` *anInteger* — Disallowed. To create a new Class or Metaclass, use `Class | subclass:instVarNames:..` instead.

Queries

`isMeta` — Returns whether the receiver is a kind of Metaclass.

Updating the Method Dictionary

`compileAccessingMethodsFor:` *anArrayOfSymbols*

Reimplemented to treat class instance variables specially. Nonmodifiable classes are made temporarily modifiable while a class instance variable is updated.

Number

Number is an abstract superclass that establishes protocol for all GemStone Smalltalk numbers. Concrete subclasses include Float, SmallInteger, and Fraction.

Superclasses Magnitude, Object

Named Instance Variables None

Instance Format Pointer, Nonindexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`denominator` (Subclass responsibility.) Returns the denominator of a Fraction representing the receiver.

`kind` Returns a Symbol from the following list:

```
#normal
#subnormal
#infinity
#zero
#signalingNaN
#quietNaN
```

The symbol tells what kind of floating-point number the receiver is. Refer to IEEE standards 754 and 854 for more information. Numbers that are not instances of a floating point class are always normal.

`numerator` (Subclass responsibility.) Returns the numerator of a Fraction representing the receiver.

`sign` Returns 1 if the receiver is greater than zero, -1 if the receiver is less than zero, and zero if the receiver is zero.

Arithmetic

`*` *aNumber*	(Subclass responsibility.) Returns the result of multiplying the receiver by *aNumber*.
`+` *aNumber*	(Subclass responsibility.) Returns the sum of the receiver and *aNumber*.
`–` *aNumber*	(Subclass responsibility.) Returns the difference between the receiver and *aNumber*.
`/` *aNumber*	(Subclass responsibility.) Returns the result of dividing the receiver by *aNumber*.

`//` *aNumber*	Divides the receiver by *aNumber*. Returns the integer quotient, with truncation toward negative infinity. For example,
	`9 // 4 = 2`
	`-9 // 4 = -3`
	`-0.9 // 0.4 = -3`
	The selector `\\` returns the remainder from this division.

`abs`	Returns a Number that is the absolute value of the receiver.
`arcCos`	Returns the arc-cosine of the receiver in radians.
`arcSin`	Returns the arc-sine of the receiver in radians.
`arcTan`	Returns the arc-tangent of the receiver in radians.
`cos`	Returns the cosine of the receiver which is treated as an angle expressed in radians.
`degreesToRadians`	Assuming the receiver represents an angle in degrees, returns the angle in radians.
`exp`	Returns e raised to the power of the receiver.
`floorLog:` *x*	Returns an Integer that is the floor of the receiver's log base *x*.
`ln`	Returns the natural logarithm of the receiver.
`log:` *x*	Returns the base *x* logarithm of the receiver.
`negated`	Returns a Number that is the negation of the receiver.

`quo:` *aNumber*
Divides the receiver by *aNumber*. Returns the integer quotient, with truncation toward zero. For example,

```
-9 quo: 4 = -2
-0.9 quo: 0.4 = -2
```

The selector `rem:` returns the remainder from this division.

`radiansToDegrees`
Assuming the receiver represents an angle in radians, returns the angle in degrees.

`raisedTo:` *aNumber*
Returns the receiver raised to the power of the argument.

`raisedToInteger:` *aNumber*
Returns the receiver raised to the power of the argument.

The argument must be an integer.

`reciprocal`
Returns 1 divided by the receiver. Generates an error if the receiver is 0.

`rem:` *aNumber*
Returns the integer remainder defined in terms of `quo:` (division of the receiver by *aNumber*, with truncation toward zero).

`sin`
Returns the sine of the receiver which is treated as an angle expressed in radians.

`sqrt`
Returns the square root of the receiver. This method currently returns a Float. This will eventually change, so that exact squares of Integers and Fractions return Integers and Fractions, for instance.

`squared`
Returns the multiplicative product of the receiver and itself.

`tan`
Returns the tangent of the receiver which is treated as an angle expressed in radians.

`\\` *aNumber*
Returns the modulo remainder defined in terms of `//`. Returns a Number with the same sign as the argument *aNumber*. For example,

```
9 \\ 4 = 1
-9 \\ 4 = 3
9 \\ -4 = -3
0.9 \\ 0.4 = 0.1
```

Comparing

`hash` Returns a SmallInteger related to the value of the receiver.

Converting

`asDecimalFloat` (Subclass responsibility.) Returns a DecimalFloat representing the receiver.

`asFloat` (Subclass responsibility.) Returns a Float representing the receiver.

`asFraction` (Subclass responsibility.) Returns a Fraction that represents the receiver.

`asInteger` Returns the integer that is closest to the receiver, on the same side of the receiver as zero is located. In particular, returns the receiver if the receiver is an integer.

`asScaledDecimal:` *scale*

Returns a ScaledDecimal representing the receiver.

`asSmallFloat` Returns a SmallFloat representing the receiver.

Flow of Control

When using any of the methods in this category, your GemStone Smalltalk code will run more efficiently when the argument is a literal block (enclosed in square brackets), rather than a variable. For more information about blocks, see the *GemStone Programming Guide*.

`downTo:` *aNumber* `by:` *stepValue* `do:` *aBlock*

Iteratively evaluates the one-argument block *aBlock*, using the block's single argument as the iteration control variable. Initially, that control variable is set to the receiver. The argument *stepValue* must be a strictly positive kind of Integer, and *aNumber* must be a kind of Integer.

The block is evaluated while the control variable is greater than or equal to *aNumber*. After each evaluation, the control variable is decremented by *stepValue*. If the receiver is less than *aNumber*, the block is not evaluated at all. Returns the receiver.

`downTo:` *aNumber* `do:` *aBlock*

Iteratively evaluates the one-argument block *aBlock*, using the block's single argument as the iteration control variable. Initially, that control variable is set to the receiver. The block is evaluated while the control variable is greater than or equal to *aNumber* (which must be a kind of Integer). After each evaluation, the control variable is decremented by 1. If the receiver is less than *aNumber*, the block is not evaluated at all. Returns the receiver.

`timesRepeat:` *aBlock*

(Reserved selector.) If the receiver is greater than zero, evaluates the zero-argument block *aBlock* the number of times represented by the receiver. (If the receiver is zero or negative, *aBlock* is not executed.).

A method which sends `timesRepeat:` will have a step point generated for the send of the `timesRepeat:` and a step point for the loop index increment and test instructions.

`to:` *aNumber* `by:` *stepValue* `do:` *aBlock*

(Reserved selector.) Iteratively evaluates the one-argument block *aBlock*, using the block's single argument as the iteration control variable. Initially, that control variable is set to the receiver. The argument *stepValue* must be a strictly positive kind of Integer, and *aNumber* must be a kind of Integer.

The block is evaluated while the control variable is less than or equal to *aNumber*. After each evaluation, the control variable is incremented by *stepValue*. If the receiver is greater than *aNumber*, the block is not evaluated at all. Returns the receiver.

A method which sends `to:by:do:` will have a step point generated for the send of the `to:by:do:` and a step point for the loop index increment and test.

`to:` *aNumber* `do:` *aBlock* (Reserved selector.) Iteratively evaluates the one-argument block *aBlock*, using the block's single argument as the iteration control variable. Initially, that control variable is set to the receiver. The block is evaluated while the control variable is less than or equal to *aNumber* (which must be a kind of Integer). After each evaluation, the control variable is incremented by 1. If the receiver is greater than *aNumber*, the block is not evaluated at all. Returns the receiver.

A method which sends `to:do:` will have a step point generated for the send of the `to:do:` and a step point for the loop index increment and test.

Formatting

`printOn:` *aStream* Appends a printable representation of the receiver to *aStream* and returns the receiver.

`printString` Returns a String whose contents are a displayable representation of the receiver.

Generality

`lessGeneralThan:` *anOperand*

Returns true if the receiver has lower generality than *anOperand*, false otherwise.

`moreGeneralThan:` *anOperand*

Returns true if the receiver has higher generality than *anOperand*, false otherwise.

Intervals

`to:` *aNumber* Returns an Interval for the numbers between the receiver and *aNumber*, where each number is the previous number plus 1.

`to:` *aNumber* `by:` *stepValue*

Returns an Interval for the numbers between the receiver and *aNumber*, where each number is the previous number plus *stepValue*.

Storing and Loading

`writeTo:` *passiveObj* Converts the receiver to its passive form and writes that information on *passiveObj*.

Testing

`even`	Returns true if the receiver is an even integer, false otherwise.
`negative`	Returns true if the receiver is less than zero, false if the receiver is zero or greater.
`odd`	Returns true if the receiver is an odd integer, false otherwise.
`positive`	Returns true if the receiver is greater than or equal to zero, false if the receiver is less than zero.
`strictlyPositive`	Returns true if the receiver is greater than zero and false if it is less than or equal to zero.

Truncation and Rounding

`ceiling`	Returns the integer that is closest to the receiver, on the same side of the receiver as positive infinity. In particular, returns the receiver if the receiver is an integer.
`floor`	Returns the integer that is closest to the receiver, on the same side of the receiver as negative infinity. In particular, returns the receiver if the receiver is an integer.
`rounded`	Returns the integer nearest in value to the receiver.
`roundTo:` *aNumber*	Returns the multiple of *aNumber* that is nearest in value to the receiver.
`truncated`	(Subclass responsibility.) Returns the integer nearest in value to the receiver, in the direction toward zero.
`truncateTo:` *aNumber*	Returns the multiple of *aNumber* that is closest to the receiver, on the same side of the receiver as zero is located. In particular, returns the receiver if the receiver is a multiple of *aNumber*.

Class Protocol

Instance Creation

`new`	Disallowed.
`new:` *anInteger*	Disallowed.

Object

Object defines the basic protocol for all objects. Every object is an instance of Object or of some subclass of Object.

Superclasses None

Named Instance Variables None

Instance Format Pointer, Nonindexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`at:` *anIndex*

Returns the value of an indexed variable in the receiver. The argument *anIndex* must not be larger than the size of the receiver, and must not be less than 1.

Generates an error if *anIndex* is not a SmallInteger or is out of bounds, or if the receiver is not indexable.

`basicAt:` *anIndex*

Returns the object at the given location in the receiver. Subclasses should not reimplement this method.

`basicPhysicalSize`

Returns the number of bytes required to represent the receiver physically. If the receiver is in special format (which implies that its representation is the same as its oop), returns zero.

This method is implemented as a primitive, for improved performance.

The `basicPhysicalSize` method returns the same result as the default implementation (in class Object) of the `physicalSize` method. It makes that default implementation available even when the `physicalSize` method is reimplemented in a subclass. The `basicPhysicalSize` method should not itself be reimplemented.

`basicSize`	Returns the number of named instance variables plus the number of indexed instance variables in the receiver. This result is equivalent to GciFetchSize(self). This method is implemented as a primitive, for improved performance.
`instVarAt:` *anIndex*	If the receiver has an instance variable at *anIndex*, returns its value. Generates an error if *anIndex* is not a SmallInteger or is out of bounds, or if the receiver has no instance variables.
`physicalSize`	Returns the number of bytes required to represent the receiver physically. If the receiver is in special format (which implies that its representation is the same as its oop), returns zero. This method is implemented as a primitive, for improved performance. This method should be reimplemented for subclasses whose instances are (or may be) a composite of component parts which are objects themselves (such as Btree nodes). Since the composite object cannot be represented independently of its components, its physical size should include that of its components. However, the component objects of collection (such as an NSC) should not be confused with its contents or elements. Elements or contained objects are in a logical relationship with the collection, whereas its components are in a physical relationship. Logically related objects can be represented and stored independently.
`segment`	Returns the Segment where the receiver is located.
`size`	Returns the number of unnamed instance variables in the receiver.
`yourself`	Returns the receiver. Often useful as the last message in a series of cascaded message sends to ensure that the expression returns a known value.

Class Membership

`changeClassTo:` *aClass*

Redefines the class of the receiver to be *aClass*. For this method to execute successfully, all of the following conditions must be true:

1. The receiver's class must have the same implementation as *aClass* (byte array, pointer array, or non-sequenceable collection).
2. Both classes must have the same number of instance variables.
3. The argument *aClass* must be a subclass of the receiver's class.
4. The constraints of the receiver's class must be the same as the constraints of *aClass*.
5. If *aClass* is a kind of Set, then the class of the receiver must also be a kind of Set. (This method cannot be used to change a Bag to a Set.)
6. The argument *aClass* must not be a kernel class for which instance creation is disallowed.

Generates an error if any of these conditions is not true.

`class`

Returns the object that is the receiver's class.

`isByteKindOf:` *aClass*

Returns true if the class of the receiver is a kind of *aClass*, and if the receiver has byte format.

`isKindOf:` *aClassHistoryOrClass*

(Optimized selector.) Returns true if the class of the receiver is identical to, or is a subclass of any class in *aClassHistoryOrClass*; otherwise, returns false.

If the *aClassHistoryOrClass* argument is actually a class rather than a class history, then this method uses the class history of the argument, instead of the class itself.

This selector is optimized by the compiler and may not be reimplemented in any subclass. This implementation is so that `perform:` will work.

`isKindOfClass:` *aClass*

Returns true if *aClass* is identical to the class of the receiver or *aClass* is a superclass of the class of the receiver.

`isMemberOf:` *aClass*
Returns true if the receiver is an instance of *aClass*, or if *aClass* is contained in the class history of the receiver's class; otherwise, returns false.

`isMemberOfClass:` *aClass*
Returns true if the receiver is an instance of *aClass*.

`species`
Returns a class similar to, or the same as, the receiver's class which can be used for containing derived copies of the receiver.

`speciesForSelect`
Returns a class similar to, or the same as, the receiver's class which can be used for containing derived copies of the receiver for select: and reject: queries.

Clustering

These methods allow you to congregate related objects in the smallest possible region, so that accessing those objects will, in general, require fewer disk accesses than would random placement. For more information about clustering, see "Clustering Objects for Faster Retrieval" in the *GemStone Programming Guide*.

`cluster`	This method clusters an object using the current default ClusterBucket. It does not force an object to disk. Rather an object not yet on disk is tagged so that the object remembers what cluster bucket is supposed to be used at such time as the object actually goes disk. If the object is large, then all nodes of the object are clustered into the same bucket. If the object is a kind of UnorderedCollection with indexes, this method does not cluster indexes. To cluster indexing objects, use the clusterIndexes method. Alternatively, see the `UnorderedCollection | clusterDepthFirst` method. No action is taken to cluster objects referenced by user-defined tags. Has no effect and returns true if the receiver was previously clustered in the current transaction; otherwise returns false after clustering the receiver. If the receiver is in SystemSegment or DataCuratorSegment, and the current session does not have write authorization for the object, this method has no effect and returns true.
`clusterBucket`	Returns the cluster bucket for the object. Returns nil if the object is a temporary object that is neither on disk nor previously clustered.

`clusterDepthFirst`

This method clusters the receiver and its instance variables in depth-first order. Clustering is performed on all of the objects that can be reached via a transitive traversal starting at the receiver. If an object is referenced by more than one clustering operation during the current transaction (that is, since the last commit or abort), it is located nearest the first reference.

This routine assumes that if we need to cluster inside this object that the object contains only named instance variables or indexed instance variables. It cannot have unordered instance variables since that would have to be a kind of Bag and this method is overridden in Collection.

Note that this implementation does not include clustering of any user-defined tags.

After clustering, returns true if the receiver is a byte object, otherwise returns false.

If the receiver is in SystemSegment or DataCuratorSegment, and the current session does not have write authorization for the object, then this method has no effect and returns true.

Has no effect and returns true if the receiver was previously clustered in the current transaction, or if the receiver is a special object.

`clusterInBucket:` *aClusterBucketOrId*

This method does not force an object to disk. Rather an object not already on disk is tagged so that the object remembers what cluster bucket is supposed to be used at such time as the object actually goes disk.

If the object is large, then all nodes of the object are clustered into the same bucket.

If the object is a kind of UnorderedCollection with indexes, this method does not cluster indexes. To cluster indexing objects, use the clusterIndexes method. Alternatively, see the `UnorderedCollection | clusterDepthFirst` method.

No action is taken to cluster objects referenced by user-defined tags.

If the receiver is in SystemSegment or DataCuratorSegment, and the current session does not have write authorization for the object, then this method has no effect and returns true.

Returns false. Has no effect and returns true if the receiver was previously clustered in the current transaction or if the object is a special object.

`moveToDisk`

Forces an object to disk if it is not already on disk. If the object has been clustered, the object is put in the cluster bucket specified by that previous clustering, otherwise the current default cluster bucket is used.

Has no effect on objects already on disk, or on special objects.

Returns the receiver.

Causes the `cluster` method to return true if sent to this object within this transaction.

`moveToDiskInBucket:` *aClusterBucketOrId*

If an object is not already on disk, forces the object to disk using the specified cluster bucket.

If the object is on disk, and is not in the specified cluster bucket, reclusters the object in the specified bucket.

The argument may be an instance of ClusterBucket, or aSmallInteger that specifies an instance of ClusterBucket, or the SmallInteger zero, which signifies the current default ClusterBucket.

Has no effect on special objects.

Returns the receiver.

Causes the `cluster` method to return true if sent to this object within this transaction.

`page`

This method returns an Integer identifying the disk page on which an object is stored. You can use this method to check your clustering methods for correctness.

The page on which an object is stored may change for any of the following reasons:

1. A clustering message is sent to the object or to another object on the same page.
2. The current transaction is aborted.
3. The object is modified.
4. Another object on the page with the object is modified.

For self-defining objects (SmallIntegers, AbstractCharacters, Booleans, UndefinedObjects), this method returns zero.

Note that this method may return nil if the receiver has not been committed to GemStone, regardless whether the receiver is referenced by a committed object.

`pageCreationTime` Returns a DateTime that is the approximate beginning of the life of the page containing the receiver.

The result represents the time that the receiver was last modified, clustered, moved to a new page by the Garbage Collector Gem, regenerated from a transaction log or full backup file during recovery or restore, whichever happened last.

When an object is modifed by a session, the resulting pageCreationTime is an approximate time of the object creation or modification, and may preceed the time at which the modification was committed.

If the receiver is not yet committed and has not yet been assigned to a page, returns the current time.

If the receiver is special, returns the value of the class instance variable timeStamp of the receiver's class.

Collection Membership

`in:` *aCollection* If there is an element of *aCollection* identical to the receiver, returns true. Otherwise, returns false.

Comparing

`=` *anObject* Returns true if the receiver and the argument have the same value. This method is defined here for identity, and is commonly reimplemented in a subclass to check for equality.

`==` *anObject* (Optimized selector.) Returns true if the receiver and the argument are the same object.

This selector is optimized by the compiler and may not be reimplemented in any subclass. This implementation is so that perform will work.

`basicIdentityHash` This method returns some Integer related to the identity of the receiver. If two objects compare identically (==) to each other, the results of sending `basicIdentityHash` to each of those objects is equal.

`equalsNoCase:` *aCharCollection*

Returns true if the receiver and the argument have the same value. Returns false otherwise.

Reimplemented in String and DoubleByteString to provide case-insensitive comparison. This implementation is the default for non-string objects.

`hash`

This method returns some Integer related to the contents of the receiver. If two objects compare equal (=) to each other, the results of sending hash to each of those objects must also be equal. Ordinarily, a class which reimplements the = method should also reimplement `hash`. Notice that an instance of class Object actually bases equality on identity, since it has no contents. Therefore, this implementation of `hash` actually returns an Integer related to the identity of the receiver.

`identityHash`

This method returns some Integer related to the identity of the receiver. If two objects compare identically (==) to each other, the results of sending `identityHash` to each of those objects will be equal.

`~=` *anObject*

Returns true if the receiver and the argument do not have the same value.

`~~` *anObject*

(Optimized selector.) Returns true if the receiver and the argument are not the same object.

This selector is optimized by the compiler and may not be reimplemented in any subclass. This implementation is so that perform will work.

Converting

`->` *anObject*

Returns an Association with the receiver as the key and the given object as the value.

`asOop`

Returns the value of the receiver's object-oriented pointer (oop) as a positive Integer. This is the receiver's unique identifier that distinguishes it from all other objects.

Copying

`copy` Returns a copy of the receiver which shares the receiver's instance variables.

`deepCopy` Returns a deep copy of the receiver. That is, if the receiver is a collection or a complex object, the copy has copies of the original's elements and any component parts. Those copies are also deep copies. In other words, deep copying is recursive.

Caution:
Use this method with care. It makes copies of all objects that can be reached from the receiver, which in some cases could be very large.

`shallowCopy` Returns a copy of the receiver with none of its components copied.

Disk Space Management

`findReferences` Searches GemStone for objects that reference the receiver, and returns an Array of any such objects. The search continues until all such objects have been found, or until the result contains 20 elements. (The method `findReferencesWithLimit:` allows you to specify an arbitrarily large limit for the result Array.)

If an object contains multiple references to the receiver, that object occurs only once in the result.

The result contains only those objects which reside within segments that the user is authorized to read. If this method encounters an object which it is not authorized to read, the final element of the result will be the String 'Read Authorization Error Encountered'.

The result contains both permanent and temporary objects. The temporary objects found may vary from run to run.

`findReferencesWithLimit:` *aSmallInt*

Returns an Array of objects in GemStone that reference the receiver. The search continues until all such objects have been found, or until the size of the result reaches the specified maximum *aSmallInt*.

The result contains both permanent and temporary objects. The temporary objects found may vary from run to run.

Note that this method may take a considerable length of time to execute, and the result may occupy a large amount of disk space. (Compare with `findReferences`, which limits the result to 20 elements.)

Error Handling

`cantPerform:` *aSelectorSymbol* `withArguments:` *anArray*

This method implements the default response when a message can't be performed with `_perform:withArguments:`. It raises the rtErrCantPerform exception.

`doesNotUnderstand:` *aMessageDescriptor*

Generates an error reporting that the receiver cannot respond to a message because no compiled method was found for the selector. The argument *aMessageDescriptor* is a two-element array. The first element is the selector that was not found and the second is an array of arguments for the message.

`halt:` *messageString* Raises an error. This method is intended for use in raising application-defined or user-defined errors.

`pause` Generates an error. You can use this method to establish breakpoints in methods, aside from any debugger breakpoints that may be set.

`shouldNotImplement:` *aSelector*

Generates an error reporting that the receiver cannot respond to *aSelector*. This is useful because sometimes a subclass should not respond to messages for which it has inherited methods from its superclass. For instance, class Set should not respond to `Object | at:`. Defining `Set | at:` with a `shouldNotImplement` error hides the ordering information of Set from users of instances of Set.

`subclassResponsibility:` *aSelector*

This is used in an abstract superclass to detect a protocol error. It generates an error indicating that a concrete subclass should have implemented this method.

Formatting

`asEUCString` Returns an EUCString that represents the receiver.

`asString` Returns a String that indicates the class of the receiver; for example, if the receiver is an instance of class Monkey, it returns a String of the form aMonkey. This method is often overridden in subclasses to provide behavior tailored to the class.

The result should not contain any formatting information. For example the following expression should evaluate to true:

```
#abc asString = String withAll: abc
```

This method is used by `Object | describe`. Thus GemStone error handling and Topaz are dependent upon this method being functional.

Must conform to rules for reimplementation of `Object | describe`.

`describe` Returns an instance of a subclass of CharacterCollection describing the receiver. This method is required by Topaz and by GemStone internal error handling. Any reimplementation must conform to the following rules:

- This method must not return nil.
- This method must return an instance of a subclass of CharacterCollection.

`printOn:` *aStream* — Puts a displayable representation of the receiver on the given stream.

`printString` — Returns a String whose contents are a displayable representation of the receiver.

Indexing Support

`addObjectToBtreesWithValues:` *anArray*

Add the receiver to the btrees of any equality indexes that it participates in, using the given array of index object/value pairs.

`removeObjectFromBtrees`

Remove the receiver from the btrees of any equality indexes that it participates in. Returns an Array of index object/value pairs.

Instance Migration

`invalidElementConstraintWhenMigratingInto:` *aCollection* `for:` *anObject*

Raises an error because the receiver could not be migrated due to one of its elements (*anObject*) not being a kind of aCollection's varying constraint. If users want to customize their migration behavior, they should override this method to return a new object that can be added to *aCollection*.

`invalidInstVarConstraintWhenMigratingInstVar:` *instVarName* `shouldBe:` *aClass*

Raises an error because the receiver could not be migrated due to having an inst var (named *instVarName*) whose value is not a kind of *aClass* (defined as the inst var constraint in the migration destination). Users should override this method to perform value conversions. When overridden, this method should return a new value of kind *aClass*.

`migrate` — Migrate the instance from its current class to its class's target class. If its class has no target class, do nothing.

`migrateFrom:` *anotherObject*

Takes information from the given object and puts it in the receiver. This message is sent to an object when its class is being migrated to another class to account for changes in a schema. Most of the work is done in `migrateFrom:instVarMap:`, which is the method that should be reimplemented in subclasses if additional work must be done.

Note: If the receiver is a kind of bag, then the receiver may have objects from *anotherObject* added to it.

`migrateFrom:` *anotherObject* `instVarMap:` *otherivi*

Takes information from the given object and puts it in the receiver. This message is sent to an object when its class is being migrated to another class to account for changes in a schema. The *otherivi* argument is a precalculated indirection table associating the receiver's instance variables with instance variables in the other object. If a table entry is 0, the other object is assumed to not have that instance variable.

This method should be augmented to perform other necessary initializations in the receiver.

Invariance

`immediateInvariant` Makes the receiver immediately invariant. (By comparison, when invariance is specified during subclass creation, instances become invariant when they are first committed.)

There is no protocol to reverse the effect of this method. However, the effect of this method can be undone by aborting the transaction if the change has not yet been committed to GemStone.

`isInvariant` Returns true if the receiver is currently invariant, false otherwise.

Message Handling

`perform:` *aSelectorSymbol* Sends the receiver the unary message indicated by the argument. The argument is the selector of the message. Generates an error if the selector is not unary.

`perform:` *aSelectorSymbol* `with:` *anObject*

Sends the receiver the message indicated by the arguments. The first argument is the keyword or binary selector of the message. The second argument is the argument of the message to be sent. Generates an error if the number of arguments expected by the selector is not 1.

`perform:` *aSelectorSymbol* `with:` *firstObject* `with:` *secondObject*

Sends the receiver the message indicated by the arguments. The first argument is the keyword selector of the message. The other arguments are the arguments of the message to be sent. Generates an error if the number of arguments expected by the selector is not 2.

`perform:` *aSelectorSymbol* `with:` *firstObject* `with:` *secondObject* `with:` *thirdObject*

Sends the receiver the message indicated by the arguments. The first argument is the keyword selector of the message. The other arguments are the arguments of the message to be sent. Generates an error if the number of arguments expected by the selector is not 3.

`perform:` *aSelectorSymbol* `withArguments:` *anArray*

Sends the receiver the message indicated by the arguments. The argument, *aSelectorSymbol*, is the keyword selector of the message. The arguments of the message are the elements of *anArray*. Generates an error if the number of arguments expected by *aSelectorSymbol* is not the same as the number of elements in *anArray*.

anArray must be an instance of Array.

Other Comparisons

`asciiLessThan:` *anObject*
: For objects that are not Characters or CharacterCollections, returns the result of an ordinary less-than compare. This method is reimplemented in Character, String, and DoubleByteString to provide comparision based on the ASCII collating order.

Queries

`isMeta`
: Returns false. This method is reimplemented in Metaclass to return true.

Storing and Loading

`basicLoadFrom:` *passiveObj*
: Reads from *passiveObj* the passive form of an object with named instance variable format. Converts the object to its active form by loading the information into the receiver.

`basicLoadFrom:` *passiveObj* `size:` *varyingSize*
: Read the structure from the given *passiveObj*, with named-instvar format. This is similar to `basicLoadFrom:`, but is used for objects whose size can not be preallocated at instantiation time (such as a Set).

`basicLoadFromOld:` *passiveObj*
: Read my structure from the given *passiveObj*.

`basicWriteTo:` *passiveObj*
: Converts the receiver to its passive form and writes that information on *passiveObj*.

`loadFrom:` *passiveObj*
: Reads from *passiveObj* the passive form of an object with named instance variable format. Converts the object to its active form by loading the information into the receiver.

`loadNamedIVsFrom:` *passiveObj*
: Reads named instance variables from the given passive object. The first instance variable should already have been parsed and be available in the *passiveObj* argument.

`loadVaryingFrom:` *passiveObj*

Reads the varying part of the receiver from the given passive object. Does not record the receiver as having been read. Does not read the receiver's named instance variables, if any.

`loadVaryingFrom:` *passiveObj* `size:` *varyingSize*

Reads the varying part of the receiver from the given passive object. Does not record the receiver as having been read. Does not read the receiver's named instance variables, if any.

`passivate`

Creates a passive description of the receiver that can be activated with an activate message to create a new object with the same value as the receiver.

For large objects or large graphs of objects, consider using this form:

```
PassiveObject passivate: anObject
  toStream: (GsFile openWrite: aFileName)
```

`shouldWriteInstVar:` *instVarName*

Returns whether the given instance variable should be written out. The default is to write out all instance variables.

`storeOn:` *stream*

Writes a string that, when evaluated, recreates a copy of the receiver to the given stream. The default is to use PassiveObjects to create the description.

`storeString`

Returns a string that, when evaluated, will recreate a copy of the receiver. The default is to use `storeOn:` to create the description.

`writeTo:` *passiveObj*

Converts the receiver to its passive form and writes that information on *passiveObj*. The argument must an instance of PassiveObject.

Storing and Loading Obsolete

`obsoleteInstVar:` *instVarName* `value:` *instVarValue*

This is a placeholder method with no default behavior. (It simply returns the receiver.) It can be reimplemented in subclasses to permit user-specified operations.

This message is sent when an instance of a class with the same name as the receiver's class is activated and it specifies a named instance variable that the receiver does not have. The instance variable name and value are sent as arguments to this method so that the instance may do something with them if it desires (such as transform old values into a new format in other instance variables).

Tag Management

`tagAt:` *tagNum*

Returns the receiver's value for the specified *tagNum* (1 or 2).

`tagAt:` *tagNum* `put:` *tagValue*

Sets the receiver's tag (*tagNum* = 1 or 2) to the specified tag value. Returns the *tagValue*.

Testing

`canBeWritten`

Returns true if the current user has write authorization for the receiver, false if not.

`isBehavior`

Returns true if the receiver is a kind of Behavior (that is, a Class or MetaClass object). Returns false otherwise.

`isCommitted`

Returns true if the receiver existed in GemStone at the time the current transaction began. Returns false otherwise.

`isConnected`

Returns true if the receiver is considered connected to GemStone. If the receiver is an uncommitted object, returns false.

However, it may return true even if the object is disconnected but not yet garbage collected. The interesting case is for temporary objects that have been forced to disk; these would be committed but not connected.

`isEquivalent:` *anObject*	Returns true if the receiver is equivalent to *anObject*. This is used to test the equivalence of characters and strings. At this level, two object are equivalent if they are identical.
`isNil`	(Reserved selector.) Returns true if the receiver is nil, false otherwise. This selector is optimized by the GemStone Smalltalk compiler and may not be reimplemented. This implementation is provided to allow execution via `perform:`.
`isSpecial`	Returns true if the receiver is a special object (that is, AbstractCharacter, Boolean, SmallInteger, or nil). Returns false otherwise.
`isSymbol`	Returns false. Classes whose instances are a kind of Symbol should reimplement this method to return true.
`isWritten`	Returns true if the receiver is an uncommitted object, or if it is a committed object that has been written since the last commit, abort, or begin transaction command was executed.
`notNil`	(Reserved selector.) Returns true if the receiver is not nil, false otherwise. This selector is optimized by the GemStone Smalltalk compiler and may not be reimplemented. This implementation is provided to allow execution via `perform:`.
`respondsTo:` *aSelector*	Returns true if the receiver's class has a method with the given selector and false if not.
`validateIsClass`	Generates an error if the receiver is not a kind of Class.
`validateIsIdentifier`	Generates an error if the receiver is not a kind of Symbol, or contains characters that are not allowed in a GemStone Smalltalk identifier.
`validateIsVariant`	Generates an error if the receiver is not variant.

Updating

`assignToSegment:` *aSegment*

Reassigns the receiver to *aSegment*. The user must be authorized to write to both segments (the receiver's current segment and *aSegment*). Generates an error if the Repository containing *aSegment* is full, if there is an authorization conflict, or if the receiver is a self-defining object (SmallInteger, AbstractCharacter, Boolean, or UndefinedObject).

`at:` *anIndex* `put:` *aValue*

Stores the argument *aValue* in the indexed variable of the receiver indicated by *anIndex*. The argument *anIndex* must not be larger than 1 + the size of the receiver, and must not be less than 1.

Generates an error if *anIndex* is not a SmallInteger or is out of bounds, if the receiver is not indexable, or if the receiver is not of the right class to store the given value.

The primitive is equivalent to GciStoreIdxOop or GciStoreByte, depending on implementation of the receiver.

`basicAt:` *anIndex* `put:` *aValue*

Puts the given object into the given location in the receiver. Subclasses should not reimplement this method.

`become:` *anObject*

Swaps the identities of the receiver and the argument.

Intended only for experienced GemStone Smalltalk programmers who need to migrate instances of one class to another.

The sender is responsible for checking the consistency of the class histories of the argument and the receiver. This method makes no such checks.

The argument, the receiver, or both are permitted to be invariant.

Neither the argument nor the receiver may be special objects (instances of classes such as SmallInteger, Character, or Boolean). Also, neither may be instances of a class that is a kind of StackSegment, StackBuffer, Activation, Process, VariableContext, or BlockClosure.

Neither the argument nor the receiver may be a kind of Bag that has indexes built on it. If either the receiver or the argument (or both) participate in an index, then either both must be in byte format or neither must be in byte format. That is, one cannot be in byte format if the other is not also. To avoid the error conditions triggered by presence of indexes, remove the indexes from the relevant NSCs prior to invoking this method.

Neither the argument nor the receiver may exist as self below the sender of a `become:` message on the active GemStone Smalltalk stack.

Once the identities have been swapped, the argument and receiver may no longer satisify the constraints of objects that reference them. This condition can lead to the failure of subsequent index creation attempts. The GemStone Smalltalk programmer is responsible for correcting broken constraints.

If either the argument or the receiver is on disk, but both are not, then the one that is not on disk is moved to disk before the identities are swapped.

Any clusterIds that belong to an object on disk remain with the object. That is, the clusterIds do not follow the identities when they are swapped.

Any tags that belong to the argument and receiver are swapped between the objects. That is, the tags do follow the identities when they are swapped.

`changeToSegment:` *segment*

Assign the receiver to the given segment. This method may be reimplemented to assign components of the receiver as well.

`instVarAt:` *anIndex* `put:` *aValue*

Stores the argument *aValue* in the instance variable indicated by *anIndex*. Generates an error if *anIndex* is not a SmallInteger or is out of bounds, or if the receiver has no instance variables.

The primitive is equivalent to GciStoreNamedOop.

`nilFields`

Sets the instance variables of the receiver to nil. This is sometimes useful as an aid to quicker garbage collection.

`size:` *anInteger*

Changes the size of the receiver to *anInteger*.

If *anInteger* is less than the current size of the receiver, the receiver is shrunk accordingly. If *anInteger* is greater than the current size of the receiver, the receiver is extended and new elements are initialized to nil.

Generates an error if *anInteger* is not a SmallInteger, or if the receiver is not indexable.

Class Protocol

Storing and Loading

`loadFrom:` *passiveObj*

Reads from *passiveObj* the passive form of an object with named instance variable format. Converts the object to its active form by loading the information into a new instance of the receiver. Returns the new instance.

OrderedCollection

An OrderedCollection is a SequenceableCollection that maintains the order of objects it contains. In GemStone, OrderedCollections are very similar to Arrays. They differ from Arrays in that they understand all of the messages that Smalltalk OrderedCollections implement.

Superclasses SequenceableCollection, Collection, Object

Named Instance Variables None

Instance Format Pointer, Indexable, Variant

Subclass Creation Allowed

Instance Protocol

Adding

`add:` *newObject* `after:` *targetObject*

Adds *newObject* to the receiver immediately following the first element that is equal to *targetObject*, and returns *newObject*. If *targetObject* is not found, reports an error.

`add:` *newObject* `afterIndex:` *anIndex*

Adds *newObject* to the receiver at the index immediately following *anIndex*.

`add:` *newObject* `before:` *targetObject*

Adds *newObject* to the receiver immediately before the first element that is equal to *targetObject*, and returns *newObject*. If *targetObject* is not found, reports an error.

`add:` *newObject* `beforeIndex:` *anIndex*

Inserts *newObject* to the receiver at the index *anIndex* and returns *newObject*.

`addAll:` *aCollection* `after:` *targetObject*

Adds each element of *aCollection* to the receiver immediately following the first element of the receiver which is equal to *targetObject*. Reports an error if *targetObject* is not found. Returns *aCollection*.

`addAll:` *aCollection* `afterIndex:` *anIndex*

Adds each element of *aCollection* to the receiver immediately after the element of the receiver at index *anIndex*. Reports an error if targetObject is not found. Returns *aCollection*.

`addAll:` *aCollection* `before:` *targetObject*

Adds each element of *aCollection* to the receiver immediately before the first element of the receiver which is equal to *targetObject*. Reports an error if *targetObject* is not found. Returns *aCollection*.

`addAll:` *aCollection* `beforeIndex:` *anIndex*

Adds each element of *aCollection* to the receiver immediately before the element of the receiver at index *anIndex*. Reports an error if targetObject is not found. Returns *aCollection*.

`addAllFirst:` *aCollection*

Inserts the given objects at the beginning of the receiver. Returns the argument, *aCollection*.

`addAllLast:` *aCollection* Appends the objects in *aCollection* to the receiver. Returns *aCollection*.

`addFirst:` *anObject* Inserts the given object at the beginning of the receiver. Returns the inserted object.

`addLast:` *newObject* Adds *newObject* as the last element of the receiver and returns *newObject*.

Comparing

`hasIdenticalContents:` *anArray*

Returns true if all of the following conditions are true:

1. The receiver and *anArray* are of the same class.
2. The two arrays are the same size.
3. The corresponding elements of the receiver and *anArray* are equal.

Returns false otherwise.

Class Protocol

Instance Creation

`new:` *size*

This method is supplied for compatibility with other implementations of Smalltalk. Its behavior differs from other Smalltalks in that it ignores the extension and returns an instance with size 0.

Other Smalltalks use an instance variable to keep track of the number of elements in what are fixed-size instances. GemStone objects are dynamically extensible and the size of the object is used to keep track of the number of elements in the collection.

PassiveObject

PassiveObject provides a means for transferring data from one GemStone repository to another that is similar to VisualWorks' Binary Object Streaming Service (BOSS) and Visual Smalltalk's Object Filer.

For a discussion of activation of passive objects that were written by GemStone version 4.1.3 or earlier, see comments in the class method initialize41ClassNames.

An instance of PassiveObject converts the form of a given GemStone object from active to passive or from passive to active. A GemStone object is called active because it can respond to messages. The object's passive form cannot respond to messages, but it can be written to a text file in a standard file system outside of GemStone. A text file is the normal intermediary storage for objects that are being transferred between GemStone repositories. PassiveObjects themselves need never be transferred or committed, and are intended to exist only within a given GemStone session.

Note:
This class provides useful protocols, but it does not represent a full or complete inter-repository data transfer facility. Not all GemStone objects can be converted into passive form. Please see the GemStone Programming Guide for more background information.

Note:
Objects referenced by the user defined tags (`tagAt:1, tagAt:2`*) of an object are not included in the default passivation of an object.*

Data transfers are ordinarily accomplished by gathering all objects to be transferred into one collection, which is then passivated and reactivated. If data is transferred piecemeal, the new repository may lose information about the connectivity of objects and produce multiple copies of an object where the original repository had only one.

Finally, class Object includes two methods, `writeTo:` and `loadFrom:`, that convert an object to and from its passive form. These methods can be reimplemented to tailor the form for any given class. The first thing any `writeTo:` method must do is to identify the class of the passivated object. It does so by sending the `writeClass:` message to the passive object that stores it. The `loadFrom:` method must send the `hasRead:` message to the passive object that loads it. It then must create a new instance of the class it finds and must read all information that was written by the `writeTo:` method.

The following discussion describes some limitations of PassiveObject in detail.

Although certain atomic objects have the same oop in any GemStone repository, most objects do not. The special case that relates some atomic objects to their oops will be ignored hereafter.

Now, the identity of a GemStone object depends upon its oop. However, when you transfer an object from one GemStone repository to another, it is not possible to guarantee that it will have the same oop; its oop in the original repository may already be used by another object in the new repository. In general, an object's oop is lost during transfer.

But the interconnectivity of objects in GemStone depends upon their oops. Objects identify their relationships to each other by their oop. To preserve interconnectivity, when aPassiveObject passivates a GemStone object, it also passivates all the other objects to which it refers, and the ones to which they refer, and so on (the transitive closure of the object). It also encodes the relationships among the objects in the transitive closure so that those relationships can be restored when the object is activated in the new repository.

However, each PassiveObject can passivate (the transitive closure of) only one object at a time. If two objects are passivated, and some objects in the transitive closure of one refer to objects in the transitive closure of the other, PassiveObject has no way to capture or encode those relationships. Upon activation, the lost interrelations between the two objects may not be evident at first because duplicate objects are created in the new repository and the same values are present. Only subsequent updates in the duplicated objects will reveal their new independence of each other. Such independence may well be unintended, a semantic anomaly in the data.

To avoid difficulties, gather all data to be transferred into one collection. Passivate the collection from the original repository, then activate it in the new repository. Connect the data to the new repository as appropriate, then remove the collection used for passivation and commit. If you must passivate two or more objects, passivate only one object in any file; two objects in a file virtually guarantee data transfer errors.

The following Characters are reserved for special use within passive objects. The special meanings do not apply within the byte contents of Strings, ByteArrays or DoubleByteStrings within a passive object:

$* denotes true
$~ denotes false
$$ next byte is an instance of Character with value 0..255
$! next two bytes are instance of Character with value 0..65535, most significant byte is first.
$. denotes nil
$# denotes _remoteNil
$/ end of named instVars within a Bag
$" prefix/suffix character used to identify instVar names
$? class prefix
$: object identifier prefix
$@ global reference prefix
$% denotes metaclass reference
$ terminates a global name or string representation of an integer
$(terminates a class name
$^ begins/ends the GemStone version header (not used prior to v5.0). When activating a PassiveObject, GemStone version 4.1.3 is assumed if no version header is found.
$& reserved for future use by GemStone Systems Inc.
$) reserved for future use by GemStone Systems Inc.
$= reserved for future use by GemStone Systems Inc.
$_ reserved for customer use

Superclasses	Object
Class Variables	Intentionally undocumented.
Named Instance Variables	Intentionally undocumented.
Instance Format	Pointer, Nonindexable, Variant
Subclass Creation	Allowed

Instance Protocol

Accessing

`oldClassMap` Returns the value of the **oldClassMap** instance variable.

`oldClassMap:` *aSymbolDictionary*
Updates the value of the **oldClassMap** instance variable.

`version` Returns the value of the **version** instance variable, which represents the version of GemStone that wrote the passivated object(s). For example, version 5.0 of GemStone has the integer value 500.

File I/O

`fromClientTextFile:` *fileName*
Sets the receiver's description string from the contents of the given text file.

`fromServerTextFile:` *fileName*
Sets the receiver's description string from the contents of the given text file.

`toClientTextFile:` *fileName*
Writes the receiver's passive description to the given text file.

`toServerTextFile:` *fileName*
Writes the receiver's passive description to the given text file.

Reading

`activate` Loads the object(s) whose representation is contained in the receiver's stream.

`hasRead:` *anObject* The given object has been instantiated but not filled out with values yet: assign it an identifier. All classes must send this message to their strObject before filling in a new instance's values.

`hasRead:` *anObject* `marker:` *marker*

For objects whose values must be at least partially read before the object can be marked as read, the `objectPositionMarker` method can be used to reserve the correct object number for the as yet uninstantiated object. The marker token returned by `objectPositionMarker` can then later be used with this method to record the instantiated object. For an example see `ExecutableBlock class | loadFrom:`.

`load:` *amount* `byteStringsInto:` *byteObj*

Loads the given number of formatted byte-sized numbers into the given byte object.

`objectPositionMarker` Reserve a place for an object that's being read but has no ID yet. The marker can be used with `hasRead:marker:` when the object has been created. An example use is activation of ExecutableBlock objects, which must read a source code string and compile it to create the block object. On writing the block, the block is assigned an object number before the string, so on reading the block back in this ordering must be maintained.

`readObject` Returns the next object from the stream.

Stream I/O

`passivate:` *anObject* `toStream:` *streamOrFile*

Passivates the given object, writing the description out to the given stream. This does not result in a state where the receiver can activate the object.

It is intended that *streamOrFile* be an instance of GsFile opened for writing, and that a new instance of PassiveObject be used to read the file when re-activation is desired.

Writing

`writeObject:` *anObject* Use this method to write components of another object to the output stream.

`writeObject:` *anObject* `named:` *objName*

Use this method to write components of another object to the output stream with instance variable names included. When read back in, the corresponding named instance variable reading method must be used.

Class Protocol

Backward Compatibility

`initialize41ClassNames`

For complete documentation you must browse this entire method.

Initialize the dictionary used to resolve class names when activating a passive object written by GemStone 4.1 . The key is a name that may be contained in a PassiveObject written by 4.1; the value is a class or object in 5.0 to be used to instantiate an object with a class name equal to that name.

Instance Creation

`fromClientTextFile:` *fileName*
Creates a new instance of the receiver, and sets the instance's description string from the contents of the given client text file.

`fromServerTextFile:` *fileName*
Creates a new instance of the receiver, and sets the instance's description string from the contents of the given text file.

`new` Disallowed. Use one of the other methods in category 'Instance Creation' to obtain a new PassiveObject.

`newWithContents:` *passiveString*
Create a new PassiveObject for a pre-existing description. This is normally used to create an instance of PassiveObject for use in activating an object previously stored into a passive textual description. The argument is the contents instance variable from a PassiveObject in which descriptions of other objects were written.

`passivate:` *anObject* Writes the given file to a new instance of this class. This method is normally invoked indirectly by sending the message passivate to an object.

Stream I/O

`newOnStream:` *aStream* Creates a new instance of the receiver, and sets the instance's description from the text on the given stream.

`passivate:` *anObject* `toStream:` *stream*
Writes the given object to the given stream, returning the stream.

PositionableStream

PositionableStream is an abstract superclass that provides additional protocol appropriate to Streams whose objects are externally named by indices. Concrete subclasses are ReadStream and WriteStream.

Superclasses Stream, Object

Named Instance Variables **itsCollection —** A SequenceableCollection; the sequence of objects that the receiver may access.

position — A SmallInteger; the current position reference for accessing the collection.

Instance Format Pointer, Nonindexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`backup` Backs up the receiver one position.

`contents` Returns the Collection associated with the receiver (that is, the sequence of objects that the receiver may access).

`next:` *count* Returns the next *count* elements in the receiver's collection.

`next:` *count* `into:` *anObject*
Stores the next *count* elements in the receiver's collection into the given object. Returns the argument *anObject*.

`nextWord` Assume that the receiver's collection is a kind of String. Returns the next word in the string or nil if there is no next word.

`peek` Returns the next element in the collection, but does not alter the current position reference. If the receiver is at the end of the collection, returns nil.

`peek2` Peeks at the second incoming object.

`peekWord` Assume that the receiver's collection is a kind of String. Returns the next word in the string without moving the receiver's position.

`skip:` *amount* — Sets the receiver's position to **position** + *amount*.

`skipAny:` *chars* — Skip past all characters in the given collection of characters. Returns the number of characters skipped.

`skipSeparators` — Skip any objects immediately next in the stream that respond true to isSeparator.

`throughAll:` *objects* — Returns a collection of objects from the receiver up to and including the sequence of objects in the argument *objects*. If the sequence of objects is not found, this returns the remaining contents of the receiver.

`upTo:` *anObject* — Returns all objects up to the given value or the end of the stream.

`upTo:` *anObject* `do:` *aBlock* — Sends each object encountered to the given block until the end of stream or the given value is encountered. Returns the receiver.

`upToAll:` *objects* — Returns a collection of objects from the receiver up to, but not including, the sequence of objects in the argument *objects*, leaving the stream positioned to read the sequence. If the sequence of objects is not found, this returns the remaining contents of the receiver.

`upToAny:` *objects* — Returns all objects up to one of the given collection of *objects* or the end of the stream.

`upToAny:` *objects* `do:` *aBlock* — Send each character encountered to the given block until the end of stream or one of the given characters is encountered.

`upToEnd` — Returns all characters from the current position to the end of the stream.

Positioning

`position`	Returns the receiver's current position reference for accessing the sequence of objects. The position is actually the next element of the collection to be read or written; the position is incremented by each read or write. In general, to start reading or writing at the nth element of a collection, the **position** must be set to n.
`position:` *anInteger*	Sets the receiver's current position reference for accessing the collection to be *anInteger*. If *anInteger* is not within the bounds of the collection, generates an error.
`reset`	Sets the receiver's position to the beginning of the sequence of objects.

Testing

`atEnd`	Returns true if the receiver cannot access any more objects, false if it can.
`isEmpty`	Returns true if the collection that the receiver accesses contains no elements; otherwise returns false.

Class Protocol

Instance Creation

`on:` *aCollection*	Returns an instance of the receiver that can stream over the elements of *aCollection*.

ProfMonitor

A ProfMonitor installs a timer to snapshot the execution stack at given intervals for a given period of time. When done monitoring, the results are collected and formed into a report showing classes, method selectors and hit rates.

Superclasses Object

Named Instance Variables

file — An instance of GsFile used to record sampling information while profiling is active. The file contains binary data in machine dependent form.

interval — The interval between sample points, in milliseconds of CPU time.

results — Holds collected, processed snapshot information in instances of a class that is private to ProfMonitor.

sampleDepth — The number of levels from top of stack to sample. Must be between 1 and 20 inclusive.

startTime — Starting cpu time (see `System | _readClock`).

endTime — Ending cpu time (see `System | _readClock`).

traceObjCreation — A Boolean. If true, object creation statistics are included in the profiling.

rawSampleArray — GemStone internal use only.

Instance Format Pointer, Nonindexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`fileName` Returns the name of the active file.

`interval:` *anInterval* Assign the sampling interval of the receiver to be *anInterval* milliseconds. *anInterval* should be a SmallInteger.

`results` Returns the value of the instance variable **results**.

Monitoring

`monitorBlock:` *aBlock*	Similar to `System	millisecondsToRun:`, this method starts profiling, executes the block, and terminates profiling.
`profileOff`	Stop the given monitor and report.
`startMonitoring`	Starts monitoring.
`stopMonitoring`	Stops monitoring.
`traceObjectCreation:` *aBoolean*	Enable (*aBoolean* == true) or disable profiling of object creation. The state change will take effect on the next invocation of ProfMonitor>>startMonitoring for the receiver.

Reporting

`gatherResults`	Analyze the receiver's file of sampling data and store the results of the analysis in the **results** instance variable of the receiver.
`report`	Formats and returns a string holding a report of the receiver's most recent profile run.
`reportDownTo:` *tally*	Formats and returns a string holding a report of the receiver's most recent profile run. Stops reporting when a tally smaller than *tally* is encountered.

Updating

`removeFile`	Removes the file generated by profiling operations in this profile monitor, if the file still exists.
`removeResults`	Releases **results** to aid garbage collection.
`results:` *newValue*	Modify the value of the instance variable **results**.

Class Protocol

Instance Creation

`new` Returns a new profiler with default initialization.

`newWithFile:` *fileName* Creates a new profiler with the given output file name and default monitoring interval.

`newWithFile:` *fileName* `interval:` *interval*
Creates a new profiler with the specified output file name and monitoring *interval* (in CPU milliseconds).

Quick Profiling

`defaultInterval` Returns the number of CPU milliseconds used for a monitoring interval if no interval is given.

`monitorBlock:` *aBlock* This is a quick way to profile the execution of a block and get a report of the result. An interval of 10 milliseconds is used, and the results are reported down to 1 hit per method. Returns a formatted report of the results of the profile run.

`monitorBlock:` *aBlock* `downTo:` *hits*
This is a quick way to profile the execution of a block and get a report of the result. Returns a formatted report of the results of the profile run.

`monitorBlock:` *aBlock* `downTo:` *hits* `interval:` *msecsPerSample*
This is a quick way to profile the execution of a block and get a report of the result. *msecsPerSample* is the CPU time interval between samples. *hits* is the minimum number of hits for a method to be included in the report.

`profileOn` Creates a default instance, starts it monitoring, and returns it. To turn off profiling, send the message profileOff to the instance.

`spyOn:` *aBlock* A convenience for Smalltalk programmers, this method merely fronts for `ProfMonitor | monitorBlock:`.

RangeIndexReadStream

RangeIndexReadStream, like its superclass BtreeReadStream, supports the composition of query results by providing access to a btree structure. Its next and atEnd methods are used the same way as those of BtreeReadStream in iterating through the btree.

RangeIndexReadStream differs from BtreeReadStream in that it uses the reverse mappings to btree nodes that are found in a RangeEqualityIndex to obtain the next entry. You can supply that index when you create the stream, and the index identifies the ordering used to return the entries.

Superclasses BtreeReadStream, Stream, Object

Named Instance Variables

rangeIndex — The RangeEqualityIndex for an instance of this class.

setIterationIndexes — An Array of Integers that indicates the offset into BucketValueBags along the path.

Instance Format Pointer, Nonindexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`next`	Returns the next value on a stream of range index values.
`rangeIndex`	Returns the value of the instance variable **rangeIndex**.
`setIterationIndexes`	Returns the value of the instance variable **setIterationIndexes**.

Copying

`copy`	Returns a copy of the receiver.

Query Support

`approxNumElements` Returns the number of leaf node entries represented by the receiver. This is approximately the number of elements because objects may share sub-objects which converge to the same leaf node entry.

`makeNsc` Returns a new NSC that contains all the elements from the receiver.

`makeNscFilterSymbols:` *bool*
Returns a new NSC that contains all the elements from the receiver. If *bool* is true, only Symbols are to be placed in the result; if *bool* is false, no Symbols are to be placed in the result.

Testing

`atEnd` Returns true if there are no more elements to return through the logical iteration of the stream.

Updating

`rangeIndex:` *newValue* Modify the value of the instance variable **rangeIndex**.

`setIterationIndexes:` *newValue*
Modify the value of the instance variable **setIterationIndexes**.

Class Protocol

Instance Creation

`new` Create an initialized instance of the receiver.

`on:` *aRangeIndex* Create a stream that can access the entire contents of the given RangeEqualityIndex.

RcCounter

Like any counter, an RcCounter maintains an integral value that can be incremented or decremented.

A single instance of RcCounter can be shared among multiple concurrent sessions without conflict. The initial value of the RcCounter at the start of a transaction in any session is the last value that has been committed. During their transactions, any or all sessions that share the RcCounter can modify it. Each session then sees a value for the RcCounter that reflects only the initial value from the start of its own transaction and the changes made in that transaction.

When a session commits the RcCounter, the cumulative changes of other transactions committed since the start of the current session's transaction are merged with those of the committing transaction. No commit conflicts occur between the sessions. But the count that a session sees immediately before its transaction is committed may not be the same as the count it sees immediately after.

Superclasses Object

Named Instance Variables None

Instance Format Pointer, Indexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`maxSessionId` Returns the maximum sessionId that can be used with this RcCounter.

`value` Returns the cumulative total of all session's counter value.

Decrementing

`decrement` Decrements the current session's counter value.

`decrementBy:` *aNumber* Decrements the current session's counter value by the given amount.

`decrementBy:` *aNumber* `ifLessThan:` *minNumber* `thenExecute:` *aBlock*
Determine if decrementing the RcCounter by the given amount would cause the total value to fall below the minimum number. If so, returns the result of executing the block; if not, performs the decrement and returns the receiver.

`decrementIfNegative:` *aBlock*
This is a convenience method to decrement the counter by one only if the counter's value does not become negative. If it would become negative, execute the Block.

Incrementing

`increment` Increments the current session's counter value by 1.

`incrementBy:` *aNumber* Increment the current session's counter value by the given amount.

Initialization

`initialize` Create subcomponents for all available session ids. This can avoid initial concurrency conflict when many sessions modify the RcCounter for the first time.

Support

`cleanupCounter` For sessions that are not logged in, centralize the individual session element's values to the global session element (at index 1). This may cause concurrency conflict if another session performs this operation.

Updating

`changeToSegment:` *segment*
Assigns the receiver and subcomponents to the given segment.

Class Protocol

Instance Creation

`new`

Returns a new RcCounter with an initial size of ten. The counter can handle four user sessions, plus the global components.

`new:` *initialNumberOfUsers*

Returns a new RcCounter with a size that supports *initialNumberOfUsers*. The new RcCounter will handle more users, but will have subcomponents created for *initialNumberOfUsers*.

RcIdentityBag

An RcIdentityBag is a special kind of IdentityBag that provides for concurrent handling of an individual instance by multiple sessions. Any or all of those sessions can modify the single instance. When that happens, RcIdentityBag also reduces the transaction conflicts that can arise among those sessions when they attempt to commit the instance to GemStone.

Superclasses IdentityBag, UnorderedCollection, Collection, Object

Named Instance Variables **components —** For GemStone internal use.

Instance Format Nsc, Nonindexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`at:` *offset*

Returns the element of the receiver that is currently located at logical position *offset*.

The elements of an RcIdentityBag are inherently unordered, and can change position (offset) when the RcIdentityBag is altered. Thus, after an RcIdentityBag is altered, a given element may reside at a different offset than before, and a given offset may house a different element. You should not infer an ordering for an RcIdentityBag's elements when you access them by offset.

This method is useful primarily as a code optimizer for iterating over all the elements of an RcIdentityBag (using a loop that runs the offset from 1 to the size of the RcIdentityBag). The RcIdentityBag must not change during the iteration. But the iteration may run faster than it would if you use other alternatives, such as the `do:` method.

`maxSessionId` Returns the maximum sessionId that can be used with this RcIdentityBag.

`size` Returns the number of elements contained in the RcIdentityBag. First checks the rc value cache, and if not there, calculate it.

Adding

`add:` *newObject* Add the object to the the RcIdentityBag. Returns *newObject*.

`add:` *newObject* `logging:` *aBoolean*
Adds the object to this session's addition bag in the RcIdentityBag. If *aBoolean* is true, logs the addition to the redo log. Returns the receiver.

`add:` *anObject* `withOccurrences:` *anInteger*
Add the object to the RcIdentityBag the given number of times. Returns the receiver.

`addAll:` *aCollection* Adds the collection of objects to the the RcIdentityBag. Returns the receiver.

`addAll:` *aCollection* `logging:` *aBoolean*
Adds the collection of objects to this sessions addition bag. If *aBoolean* is true, logs the addition to the redo log. Returns the receiver.

Clustering

`cluster` Clusters an object using the current default ClusterBucket. Has no effect and returns true if the receiver was previously clustered in the current transaction; otherwise returns false after clustering the receiver.

`clusterDepthFirst` Clusters the receiver and its contents in depth-first order. Returns true if the receiver has already been clustered during the current transaction; returns false otherwise. This operation may cause concurrency conflicts with other sessions.

Comparing

`=` *anRcIdentityBag*	Verifies that the receiver and *anRcIdentityBag* are of the same class, then uses the semantics of IdentityBag comparison for the elements in the RcIdentityBag.
`hash`	Returns an Integer hash code for the receiver. *Warning:* *This is a computationally expensive operation.*

Converting

`asIdentityBag`	Returns a bag that consists of objects in this RcIdentityBag. Note that each invocation of this message returns a new bag.

Copying

`copy`	Returns a copy of the receiver. A copy must not include any indexes that exist on the receiver.

Enumerating

`do:` *aBlock*	Enumerates over all elements in the RcIdentityBag, executing *aBlock* with each element as the argument. Returns the receiver.

Initialization

`initialize:` *aSize*	Initializes a new instance.
`initializeComponents`	Create subcomponents for all available session ids. This can avoid initial concurrency conflict when many sessions add an object to the RcIdentityBag for the first time.

Query Support

`speciesForSelect`	Returns the class to use to select and reject queries.

Removing

`remove:` *anObject*
Removes *anObject* from the receiver. If *anObject* is present several times in the receiver, only one occurrence is removed. Generates an error if *anObject* is not in the receiver.

`remove:` *anObject* `ifAbsent:` *aBlock*
Removes *anObject* from the receiver. If *anObject* is present several times in the receiver, only one occurrence is removed. If *anObject* is not in the receiver, this method evaluates *aBlock* and returns its value. The argument *aBlock* must be a zero-argument block.

`removeAll:` *aCollection*
Removes one occurrence of each element of *aCollection* from the receiver and returns the receiver. Generates an error if any element of *aCollection* is not present in the receiver.

`removeAllPresent:` *aCollection*
Removes from the receiver one occurrence of each element of *aCollection* that is also an element of the receiver. Differs from `removeAll:` in that, if some elements of *aCollection* are not present in the receiver, no error is generated. Returns the receiver.

`removeIfPresent:` *anObject*
Remove *anObject* from the receiver. If *anObject* is present several times in the receiver, only one occurrence is removed. Returns nil if *anObject* is missing from the receiver.

Searching

`includes:` *anObject*
Returns true if *anObject* is present in any session component of the RcIdentityBag.

`includesIdentical:` *anObject*
Returns true if *anObject* is present in any session component of the RcIdentityBag.

`includesValue:` *anObject*
Returns true if *anObject* is present in any session component of the RcIdentityBag. Uses equality comparison.

`occurrencesOf:` *anObject*

Returns the number of occurrences of *anObject* in all session components of the RcIdentityBag.

`speciesForCollect` Returns a class, an instance of which should be used as the result of collect: or other projections applied to the receiver. For RcIdentityBags, uses an unconstrained RcIdentityBag for the result.

Set Arithmetic

`*` *anIdentityBag* Intersection. Returns a kind of IdentityBag containing only the elements that are present in both the receiver and the argument.

The class of the result is the class of the argument. If the argument is a kind of RcIdentityBag, the result is an IdentityBag.

If the result is a kind of IdentitySet, then each element that occurs in both the receiver and the argument occurs exactly once in the result. If the result is an IdentityBag and if an element occurs m times in the receiver and n times in the argument, then the result contains the lesser of m or n occurrences of that element.

`+` *anIdentityBag* Union. Returns a kind of IdentityBag containing exactly the elements that are present in either the receiver or the argument.

The class of the result is the class of the argument. If the argument is a kind of RcIdentityBag, the result is an IdentityBag.

If the result is a kind of IdentitySet, then each element that occurs in either the receiver or the argument occurs exactly once in the result. If the result is a kind of IdentityBag, and if an element occurs m times in the receiver and n times in the argument, then the result contains m + n occurrences of that element.

– *anIdentityBag* Difference. Returns an IdentityBag containing exactly those elements of the receiver that have a greater number of occurrences in the receiver than in the argument. If an element occurs m times in the receiver and n times in the argument (where m >= n), then the result will contain m - n occurrences of that element.

Sorting

`sortAscending:` *aSortSpec*

Returns an Array containing the elements of the receiver, sorted in ascending order, as determined by the values of the instance variables represented by *aSortSpec*.

`sortDescending:` *aSortSpec*

Returns an Array containing the elements of the receiver, sorted in descending order, as determined by the values of the instance variables represented by *aSortSpec*.

`sortWith:` *aSortPairArray*

Returns an Array containing the elements of the receiver, sorted according to the contents of *aSortPairArray*.

Storing and Loading

`basicWriteTo:` *passiveObj*

Converts the receiver to its passive form and writes that information on *passiveObj*.

Support

`centralizeSessionElements`

Place the elements of inactive session components in the global bag. This may cause concurrency conflict if another session performs this operation, or if a new session modifies the RcIdentityBag.

`cleanupBag` Iterate through all inactive session bags and process their removal bags. This may cause conflict if a new session modifies the receiver.

`distributeSessionElements`

Distributes the elements of inactive session components across all inactive session components. This could lessen the chance of conflicts that must be resolved at commit time. This may cause concurrency conflict if another session performs this operation or if a new session modifies the receiver.

Updating

`changeToSegment:` *segment*

Assigns the reciever and its subcomponents to the given segment.

Class Protocol

Instance Creation

`new` Returns a new RcIdentityBag with an initial size of 10 (can handle 4 user sessions, plus the global components).

`new:` *initialNumberOfUsers*

Returns a new RcIdentityBag with a size that supports *initialNumberOfUsers*. The new RcIdentityBag will handle more users, but will have subcomponents created for *initialNumberOfUsers*.

RcKeyValueDictionary

RcKeyValueDictionary is an AbstractDictionary that shares many of the protocols and characteristics of KeyValueDictionary. Like all dictionaries, it stores key/value pairs. In an RcKeyValueDictionary, keys may be of mixed classes.

Like KeyValueDictionary, RcKeyValueDictionary stores key/value pairs under an index that is generated by applying a hash function to the key; it does not use Associations. The hashing improves retrieval speed. However, you must observe an important restriction: after a key/value pair has been added to an RcKeyValueDictionary, you must not modify the key. Doing so renders the value inaccessible.

An RcKeyValueDictionary is also an equality-based collection. That is, two keys or two values are considered to be the same if they are equivalent; they need not be identical to be the same. Thus, if you add two key-value pairs to an RcKeyValueDictionary but the keys are equivalent, even if they are not identical, then the result is that the second pair overwrites the first one, because the keys are the same.

However, unlike KeyValueDictionary, RcKeyValueDictionary provides for concurrent handling of an individual instance by multiple sessions. Any or all of those sessions can modify the single instance. When that happens, RcKeyValueDictionary also reduces (but does not eliminate) the transaction conflicts that can arise among those sessions when they attempt to commit the instance to GemStone.

Commit Conflicts. In general, RcKeyValueDictionaries do not cause concurrency conflicts for write operations that are commutative (operations that can be performed in any order without affecting the final GemStone state). However, under some circumstances a user may experience conflict for commutative operations when the basicSize of the dictionary is too small (relative to the number of write operations performed in a transaction). This can be avoided by creating a larger dictionary with the `new:` method, or increasing an existing dictionary's size with the `rebuildTable:` method.

If multiple users change values for different keys in a single RcKeyValueDictionary, the changes do not usually cause conflicts at commit time. However, there is a (narrow and uncommon) window of time over which users have no control during which such a set of changes could result in conflicts.

Superclasses	AbstractDictionary, Collection, Object
Named Instance Variables	**collisionLimitPerBucket —** A SmallInteger that represents the number of collisions allowed in a bucket before rebuilding the hash table.
Instance Format	Pointer, Indexable, Variant
Subclass Creation	Allowed

Instance Protocol

Accessing

`at:` *aKey* `ifAbsent:` *aBlock*

Returns the value that corresponds to *aKey*. If no such key/value pair exists, returns the result of evaluating the zero-argument block *aBlock*.

`at:` *aKey* `otherwise:` *aValue*

Returns the value that corresponds to *aKey*. If no such key/value pair exists, returns the given alternate value.

`keys` Returns an IdentitySet containing the receiver's keys.

`numElements` Same as size.

`size` Returns the number of key/value pairs in the receiver.

`values` Returns an OrderedCollection containing the receiver's values.

Clustering

`clusterDepthFirst` This method clusters the receiver and its named instance variables and the key/value pairs in depth-first order. Returns true if the receiver has already been clustered during the current transaction; returns false otherwise.

Comparing

= *aKeyValueDictionary*

Compares two RcKeyValueDictionaries for equality. Returns true if all of the following conditions are true:

1. the receiver and *aKeyValueDictionary* are of the same class,
2. the two RcKeyValueDictionaries have the same number of elements
3. all of the keys in one dictionary return the same value in both dictionaries.

Returns false otherwise.

`hash`

Since `RcKeyValueDictionary | =` is based on identity of elements, hash is based on identityHash of elements. Returns the numerical hash value.

Copying

`copy`

Copies the collision buckets and returns a copy of the receiver.

Enumerating

`keysAndValuesDo:` *aBlock*

Iteratively evaluates the two argument block, *aBlock*, using each key and value of the receiver as the argument to the block. Returns the receiver.

Hashing

`rebuildTable:` *newSize*

Rebuilds the hash table by saving the current state, initializing and changing the size of the table, and adding the key value pairs saved back to the hash dictionary. This method is intended to be used internally. If it is invoked directly by an application, concurrency conflicts may result.

Initializing

`initialize:` *itsSize*
Initializes the instance variables of the receiver to be an empty RcKeyValueDictionary of the specified size. This is intended to be used internally at the time of instance creation. If used on an existing, populated RcKeyValueDictionary, concurrency conflicts may result.

`tableSize:` *aSize*
Sets the size of the receiver to a new value. Each collision bucket is reset to contain no elements. This method is intended to be used internally. If it is invoked directly by an application, concurrency conflicts may result.

Statistics

`statistics`
Returns a Dictionary containing statisitics that can be useful in determining the performance of a key-value dictionary, including the following information:

- TotalCollisionBuckets: The number of collision buckets required to implement the key-value dictionary.
- AvgPairsPerBucket: The average number of key/value pairs in each bucket.
- LargestBucket: The bucket having the most key/value pairs. This bucket contains the most keys for which the hash function did not provide a good distribution over the range of values in the table.

Since RcKeyValueDictionaries are implemented differently than KeyValueDictionaries, the statistics are calculated as if the implementation were the same as KeyValueDictionaries. To be specific, RcKeyValueDictionaries are implemented with all collision buckets. Therefore, the calculations don't count CollisionBuckets with a size of 1 as a collision.

Updating

`changeToSegment:` *segment*
Assigns the receiver and any collision buckets to the given segment.

Class Protocol

Accessing the Class Format

`firstPublicInstVar` Returns the index of the first publicly available instance variable storage location, whether or not a public instance variable has actually been defined.

Instance Creation

`new` Returns a RcKeyValueDictionary with the default table size.

`new:` *tableSize* Returns a RcKeyValueDictionary with the specified table size.

RcQueue

An RcQueue (reduced-conflict queue) is an implementation of a FIFO queue that provides significantly reduced concurrency conflicts when used in an environment with multiple producers (users that add elements to the queue) and a single consumer (a user thay removes items from the queue). Producers are guaranteed not to conflict with each other, nor with a single consumer. An RcQueue is implemented as a collection of RcQueueSessionComponents, each of which contains the queue elements submitted by a particular session.

When there is a conflict on an RcQueue that prevents a transaction from committing successfully, the state of the RcQueue is updated to include modifications from other transactions, and the modifications of the current transaction are lost. In addition, other RcQueues that were modified in the current transaction may also lose their modifications if they experienced physical conflicts (even if the conflicts were not logical conflicts). This situation is avoided entirely if RcQueues are used in their intended manner (single consumer, multiple producers).

Subclassing. RcQueue employs lazy initialization of its elements; they are initialized only when needed. If you create a subclass of RcQueue, your code must check that an element is not nil before it is used. Reimplementations of methods such as `add:`, `remove:`, and `do:` are especially sensitive.

Superclasses	Collection, Object
Named Instance Variables	**removalSeqNumbers —** An instance of the class RcQueueRemovalSeqNumbers (essentially an Array of SmallIntegers) representing the order in which elements are to be removed from the queue.
Instance Format	Pointer, Indexable, Variant
Subclass Creation	Allowed

Instance Protocol

Accessing

`maxSessionId`	Returns the max sessionId that can be used with this RcQueue.
`numberInvalid`	Returns the number of entries in the RcQueue which have been removed but not yet reclaimed in the SessionComponents for sessions that are not active. The intent is to determine if it is worth invoking cleanup.
`size`	Returns the number of valid entries in the RcQueue.

Adding

`add:` *aValue*	Adds *aValue* to the RcQueue, returns the receiver.

Clustering

`clusterDepthFirst`	Performs no action, as clustering defeats the conflict-reduction scheme.

Enumerating

`do:` *aBlock*	Evaluates *aBlock* with each of the current elements of the RcQueue as the argument. The argument *aBlock* must be a one-argument block. This method does not traverse the queue elements in order.

Initialization

`initialize`	Create subcomponents for all available session ids. This can avoid initial concurrency conflict when many sessions add an object to the RcQueue for the first time.

Performance Enhancement

`cleanupMySession`	Cleans up the entries for my session that have already been removed by the consumer. This method is only needed if a producer adds entries to the queue faster than the consumer removes them and then no longer adds to the queue. In this situation, the producer can enhance the performance of the consumer by either logging out or periodically executing this method and committing.

Removing

`peek` Returns the leading element from the receiver without removing it. If the receiver is empty, returns nil.

`remove` Removes the leading element from the receiver and returns that element. If the receiver is empty, returns nil.

`removeAll` Removes all entries from the RcQueue, and returns an Array that contains those entries, in order. It is more efficient to use `removeAll` than to send the message `remove` repeatedly.

`removeCount:` *maxToRemove*
Removes entries from the RcQueue, and returns an Array that contains the minimum of *maxToRemove* or the queue size entries, in order. It is more efficient to remove multiple entries than to send the message `remove` repeatedly.

Searching

Searching methods are disallowed because (a) creating a copy of RcQueue containing a subset of the queue's elements defeats the conflict-reduction scheme and (b) accessing a queue's elements in an order other than First-In/First-Out is contrary to the purpose of a queue. Consider using `removeAll`, which removes all elements from the queue and stores them in an Array.

If your application must operate on all of the entries in an RcQueue without removing them in an orderly fashion, you can use `do:` to enumerate the elements of the RcQueue. Be aware, however, that the `do:` method does not traverse the queue in order.

`collect:` *aBlock* Disallowed. Use `do:` to enumerate the elements of an RcQueue. It doesn't make much sense to build another RcQueue with a portion of its contents.

`detect:` *aBlock* Disallowed. Use `do:` to enumerate the elements of an RcQueue.

`detect:` *aBlock* `ifNone:` *exceptionBlock*
Disallowed. Use `do:` to enumerate the elements of an RcQueue.

`includes:` *anObject* Disallowed. Use `do:` to enumerate the elements of an RcQueue.

`includesIdentical:` *anObject*
Disallowed. Use `do:` to enumerate the elements of an RcQueue.

`includesValue:` *anObject*
Disallowed. Use `do:` to enumerate the elements of an RcQueue.

`occurrencesOf:` *anObject*
Disallowed. Use `do:` to enumerate the elements of an RcQueue.

`reject:` *aBlock* Disallowed. Use `do:` to enumerate the elements of an RcQueue.

`select:` *aBlock* Disallowed. Use `do:` to enumerate the elements of an RcQueue.

`speciesForCollect` Returns a class, an instance of which should be used as the result of `collect:` or other projections applied to the receiver.

Testing

`isEmpty` Returns true if the queue is empty, and false otherwise.

Updating

`changeMaxSessionId:` *newMaxSessionId*
Changes the maximum number of sessions for which the RcQueue is configured. Modifying the capacity of an RcQueue in this way may cause concurrency conflicts with the consumer session, if one is active.

`changeToSegment:` *segment*
Assigns the receiver and subcomponents to the given segment.

`cleanupQueue` Removes obsolete entries belonging to inactive sessions. Can cause concurrency conflicts with the consumer.

`size:` *anInteger* Disallowed. You cannot change the size of anRcQueue other than to add or remove elements. To change the maximum sessionId that can be used with this queue see `changeMaxSessionId:`.

Class Protocol

Instance Creation

`new` Returns a new RcQueue.

`new:` *initialNumberOfUsers*

Returns a new RcQueue with a size that supports *initialNumberOfUsers*. The new RcQueue will handle more users, but will have subcomponents created for *initialNumberOfUsers*.

ReadStream

A ReadStream is a PositionableStream that allows its objects to be read but not written.

Superclasses PositionableStream, Stream, Object

Named Instance Variables None

Instance Format Pointer, Nonindexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`next`
Returns the next object that the receiver can access for reading. Generates an error if an attempt is made to read beyond the end of the stream.

`nextElements:` *count* `into:` *anObject*
Stores the next *count* elements that the receiver can access for reading into *anObject*. The receiver's collection and *anObject* must be compatible SequencableCollections. Returns the count of elements read.

Generates an error if an attempt is made to read beyond the end of the stream.

Adding

`nextPut:` *anObject*
Disallowed. You cannot write to a ReadStream.

Class Protocol

Instance Creation

`new`
Disallowed. To create a new ReadStream, use the class method on: instead.

Repository

A Repository is an object that represents a virtual storage into which users can place their data. Each Repository is an Array of up to 4096 Segments, some of which may be nil. Repositories are described in the *GemStone Programming Guide*.

Superclasses Array, SequenceableCollection, Collection, Object

Named Instance Variables

name — A Symbol; the user-supplied logical name for the Repository.

dataDictionary — A dictionary that describes the logical entry points to data in this Repository; a convenient mechanism for remembering and accessing the objects in the Repository, similar to a file system directory.

Instance Format Pointer, Indexable, Variant

Subclass Creation Disallowed

Instance Protocol

Accessing

`dataDictionary` Accesses the user-defined data dictionary. (Not for use in this product release.)

`fileNames` Returns an Array of Arrays containing the filenames for the extents and replicates of the receiver.

Each Array within the returned Array contains two Strings. The first String represents the filename of the Nth extent (where N is the index into the returned Array). The second element represents the filename of the replicate of the Nth extent. If there is no replicate, the 2nd element is a zero-length String.

`name` Returns the logical name of the receiver (a Symbol).

Adding

`add:` *newObject* Disallowed.

`addAll:` *aCollection* Disallowed.

`addLast:` *newObject* Disallowed.

`insert:` *aSequenceableCollection* `at:` *anIndex*
Disallowed.

`insertAll:` *aSequenceableCollection* `at:` *anIndex*
Disallowed.

Backup and Restore

Backups and restoration are ordinarily performed while using the GemStone DataCurator login. It is possible to use another login that also has the FileControl privilege. However, for restorations, it is recommended that you use only the DataCurator or the SystemUser logins. If you use another login, and that login disappears as a result of the restoration, you will see a fatal error.

The *GemStone System Administration Guide* discusses backups and restoration in more detail.

`abortFullBackup` Cancel a full backup that is in progress. If `fullBackupTo:` has been used to start a multi-file backup, but `continueFullBackupTo:MBytes:` has not been executed to completion of the backup, you can use `abortFullBackup` to cancel the full backup and permit this session to commit and abort.

This method requires no privileges. If a backup is not in progress, this has no effect.

`abortRestore`

If a restore from backups is in progress on multi-file backup, this method cancels the restore, and reenables logins. The Repository reverts to the state prior to starting the restore.

If the last file of a backup has already been restored, or a restore from a multifile backup is not in progress, this method has no effect.

Note that this method has no effect if a restore from backup has completed and restore from transaction logs is in progress. To stop restoring transaction logs you must use `commitRestore`.

Returns true.

This method requires the FileControl privilege. It is recommended that you run as either DataCurator or SystemUser.

If a backup file read with `restoreFromBackup:` is truncated or corrupt, it may be necessary to execute `abortRestore` before `restoreFromBackup:` can be used to restart the restore from a good copy of the backup file.

`commitRestore`

Terminates a restore operation and permits normal commits. Returns true if the commit of the restores succeeded. Otherwise, either returns a String describing a warning or generates an error.

The restore operation must have been started with `restoreFromBackup:`. Otherwise, this method generates an error.

If `restoreFromCurrentLogs` was not the immediately preceding restore operation, then a warning is issued, but the termination of restore will succeed. Such use of `commitRestore` can result in failure to restore all previously committed transactions. However, this allows a Repository to be restored as far as practical when some log files are lost or corrupted.

If GemStone was using partial-logging mode at the time restored backup file(s) were written, or if GemStone is currently in partial-logging mode (STN_TRAN_FULL_LOGGING is false in the stone's configuration file), then `commitRestore` is not needed, since the last `restoreFromBackup:` will have committed the restore.

You must be the only user logged in, otherwise an error is generated. The `restoreFromBackup:` that started the restore process will have suspended logins, and a successful `commitRestore` will reenable logins.

The session is put into manualBegin transaction mode, and is left outside of a transaction after this method executes.

This method requires the FileControl privilege. It is recommended that you run as either DataCurator or SystemUser.

`continueFullBackupTo:` *fileOrDevice* `MBytes:` *mByteLimit*

Continue a full backup by writing a second or subsequent backup file as specified by *fileOrDevice*, with a size limit specified by *mByteLimit*.

This method operates outside of a transaction, and leaves the session outside of a transaction. The session may do one or more aborts during the execution of the backup to avoid causing excessive repository growth.

See `fullBackupTo:MBytes:` for additional description of the arguments.

Returns true if the backup was completed. Returns a message (a String) if `continueFullBackupTo:MBytes:` should be run to complete the backup.

This method requires the FileControl privilege.

A GciHardBreak during this method will terminate the session.

`fullBackupTo:` *fileOrDevice*

Backup the receiver to a single backup file or tape. See `fullBackupTo:MBytes:` for further documentation.

This method requires the FileControl privilege.

A GciHardBreak during this method will terminate the session.

`fullBackupTo:` *fileOrDevice* `MBytes:` *mByteLimit*

Produces a full backup file containing the most recently committed version of the receiver as of the time the method is executed.

The argument *fileOrDevice* (a kind of String) specifies the file or device where the backup is to be created.

If *fileOrDevice* does not specify a file on some file system, then it may be a device name specifying either a raw disk partition or a tape device. The *fileOrDevice* argument may use GemStone Network Resource String syntax. For example, this may be used to access a tape device on another machine, provided a GemStone netldi process is running on the remote machine.

If *fileOrDevice* specifies a file that already exists on a fileSystem, or if it specifies a raw disk partition that already contains a GemStone extent, transaction log, or backup file, then an error is generated. Use the removedbf utility to erase raw disk partitions.

The *mByteLimit* argument, which specifies the maximum size of *fileOrDevice* in units of megabytes, must be a SmallInteger. The value 0 means that there is no limit on the size of the resulting *fileOrDevice*. A *mByteLimit* less than 0 or greater than 4096000 will generate an error.

If the backup requires more bytes than you specified in *mByteLimit*, this method returns a message (a String) stating that a partial backup file was created. In this case, further commits and aborts in this session are disallowed until you either complete the full backup with `continueFullBackupTo:MBytes:` or cancel the backup with `abortFullBackup`. To continue the backup, you can execute the method

`continueFullBackupTo:MBytes:`, which creates the next file in the backup sequence.

If *fileOrDevice* runs out of space, such as off the end of a tape, the backup will terminate with a system I/O error at that point. The backup will be unusable. To avoid having to repeat the entire backup, make sure the device has sufficient space or set *mByteLimit* appropriately.

When the size of your GemStone repository exceeds the capacity of a backup tape, file system, or raw disk partition, you can use *mByteLimit* (a SmallInteger) to control the maximum number of bytes to be written to the backup file.

This method always puts the session into auto-begin transaction mode, aborts the current transaction, and then commits a record of the start of the backup to `UserGlobals at: #BackupLog`. This commit is done as a checkpoint. Then the transaction mode is changed to manual begin, and the remainder of the backup operation executes outside of a transaction so that it does not cause excessive repository growth. A varying number of aborts are done while outside of a transaction, depending on the time required to execute the backup.

When the backup completes, the session is always left outside of a transaction so that it does not retain a commit record that would cause the repository to grow.

Returns true if the backup was completed. Returns a message (a String) if `continueFullBackupTo:MBytes:` should be run to complete the backup.

This method requires the FileControl privilege.

A GciHardBreak during this method will terminate the session.

`restoreFromArchiveLogDirectories:` *arrayOfDirectorySpecs*
`tranlogPrefix:` *tranlogPrefixString*
`replicateDirectories:` *arrayOfReplicateDirSpecs*
`replicatePrefix:` *replicPrefixString*

This method is equivalent to invoking `setArchiveLogDirectories:tranlogPrefix:replicateDirectories:replicatePrefix:` followed by `restoreFromArchiveLogs`.

Please see those two methods for complete descriptions.

`restoreFromArchiveLogs`

Given a Repository already in restore mode from a previous restore operation, restores all available tranlogs contained in the directories specified by the last preceding invocation of either

```
Repository>>
   setArchiveLogDirectories:...
      replicatePrefix:
```

or

```
Repository>>
   restoreFromArchiveLogDirectories:...
      replicatePrefix:
```

Determines the restore status's current fileId by doing the equivalent of `SystemRepository restoreStatus`. Then attempts to restore contents of any log file whose current fileId is beyond the end of the last restore, if the log file can be found when searching the directories previously specified.

Generates an error if neither setArchiveLogDirectories:... nor restoreFromArchiveLogDirectories:... has been executed since the last startstone of this Repository.

When executed using Topaz, the result is either a String describing the success of the operation (in which case the Topaz result (obj **) may be nil), or an error message.

This method terminates GemStone Smalltalk execution and does an automatic abort. All GemStone Smalltalk temporary objects present at the start of this method are destroyed by this method, so it can only be executed from Topaz.

You must be the only user logged in, otherwise an error is generated.

This method requires the FileControl privilege. It is recommended that you run as either DataCurator or SystemUser. This method puts the session into manualBegin transaction mode.

Note that restore status is an attribute of the Repository, not of a session, so the required preceding restore operation could have been executed in some preceding session.

`restoreFromBackup:` *fileOrDevice*

If a restore is not in progress, starts a full restore of the receiver by initializing a shadow object space and reading the first backup file into that space. Normal commits are disallowed while a restore is in progress.

If a restore is in progress, continues the restore by reading a second or subsequent backup file from a multi-file backup set.

Use the method `restoreStatus` to determine whether a restore is in progress or not, and the next file expected in a multiple file restore. Use the method `abortRestore` to cancel a restore that stopped prematurely due to *fileOrDevice* being truncated or corrupt, before attempting the restore with a good copy of *fileOrDevice*.

If the *fileOrDevice* is the last backup file in a backup set, the shadow object space is automatically made visible to GemStone Smalltalk at the completion of this method, and if GemStone was in full-logging mode at the time of the backup, the object server is made ready for `restoreFromLog:`. This installation of the restored object table terminates GemStone Smalltalk execution and does an automatic abort. All GemStone Smalltalk temporary objects present at the start of this method are

destroyed, so this method can only be executed from Topaz. Once the last file backup file in a backup set has been restored, the restore status of the Repository will persist across sessions and shutdowns of the stone.

If the *fileOrDevice* is the last backup file in a backup set and GemStone was in partial logging mode at the time of the backup, then the Repository is ready for normal use after the restore of the file.

If the last file of a backup set has not yet been restored, the shadow object space is thrown away if this session logs out, or if `abortRestore` is executed. After a fresh login the restore would have to be restarted with the first backup file again.

When executed using Topaz, the result is either a String describing the success of the operation (in which case the Topaz result (obj **) may be nil), or an error message.

The backup file must have been previously created with one of the following: `fullBackupTo:MBytes:`, `fullBackupTo:`, or `continueFullBackupTo:MBytes:`.

Restored objects will be clustered in a manner that is similar, but not necessarily identical, to the clustering organization at the time the backup file was created.

If the Repository being restored into has the same number of extents as the Repository from which the full backup was made, then distribution of objects within extents is preserved. In this case the DBF_ALLOCATION_MODE configuration parameter is ignored during the restore, unless an extent hits a size limit specified by DBF_EXTENT_SIZES. If the number of extents differs, then the DBF_ALLOCATION_MODE configuration parameter at the time of the restore will control distribution of objects across the extents. The number of extents recorded in the backup file is the number of extents as of the start of the full backup.

You must be the only user logged in, otherwise an error is generated. This method suspends logins. Logins will be reenabled when one of the following occurs:

1. this session logs out.
2. the last backup files(s) of a backup have been restored, and the backup was made when in partial logging mode.
3. `commitRestore` succeeds.

It is recommended that the stone be restarted on a copy of the initial Repository, $GEMSTONE/bin/*.dbf, before executing this method, in order to minimize the size of the restored Repository.

The Garbage Collector session is shut down at the beginning of this method. If this method succeeds, then the Garbage Collector session remains shut down until `restoreFromCurrentLogs` has been successfully executed, otherwise the Garbage Collector session may be scheduled for restarting.

This method requires the FileControl privilege. It is recommended that you run as either DataCurator or SystemUser.

A GciHardBreak during this method will terminate the session.

If the session is using a shared page cache, then the async I/O function of the stone's pgsvrmain process is made more aggressive. The following settings are automatically active for the duration of this method:

```
System configurationAt:#StnMntMaxAioRate
       put: 1000.

System configurationAt:
                 #ShrPcTargetPercentDirty
       put: 5.
```

`restoreFromBackups:` *arrayOfFilesOrDevices*

Restore multiple backup files. Equivalent to executing `restoreFromBackup:` once for each element of *arrayOfFilesOrDevices*. The *arrayOfFilesOrDevices* argument must be an Array not larger than 200 elements.

When executed using Topaz, the result is either a String describing the success of the operation (in which case the Topaz result (obj **) may be nil), or an error message.

This method requires the FileControl privilege. It is recommended that you run as either DataCurator or SystemUser.

A GciHardBreak during this method terminates the session.

See `restoreFromBackup:` for further documentation.

`restoreFromCurrentLogs`

After a `restoreFromBackup:` returns true, this method may be executed to redo transactions which occurred since the backup was made. This method re-does all transactions contained in the log files that are in stone's current log directories or devices as defined by the STN_TRAN_LOG_DIRECTORIES and STN_REPL_TRAN_LOG_DIRECTORIES configuration file parameters.

When executed using Topaz, the result is either a String describing the success of the operation (in which case the Topaz result (obj **) may be nil), or an error message.

This method terminates GemStone Smalltalk execution and does an automatic abort. All GemStone Smalltalk temporary objects present at the start of this method are destroyed by this method, so it can only be executed from Topaz.

If some log files written since the restored backup file(s) were generated are no longer online, those off-line logs must be processed using the method `restoreFromLog:` before this method can be used.

If GemStone was using partial-logging mode at the time restored backup file(s) were written then `restoreFromCurrentLogs` is not allowed.

You must be the only user logged in, otherwise an error is generated.

This method requires the FileControl privilege. It is recommended that you run as either DataCurator or SystemUser. This method puts the session into manualBegin transaction mode.

Note that restore status is an attribute of the Repository, not of a session, so the required preceding `restoreFromBackup:` could have been executed in some preceding session.

`restoreFromLog:` *fileOrDevice*

After a `restoreFromBackup:` returns true, this method may be executed to redo transactions which occurred since the backup was made. This method re-does all transactions contained in the specified log file.

When executed using Topaz, the result is either a String describing the success of the operation (in which case the Topaz result (obj **) may be nil), or an error message.

This method terminates GemStone Smalltalk execution and does an automatic abort. All GemStone Smalltalk temporary objects present at the start of this method are destroyed by this method, so it can only be executed from Topaz.

Log files must be restored in time-sequence starting from the log file that was active at the time backup was made.

Use the `restoreStatus` method to determine the next file required for a restore operation. Note that restore status is an attribute of the Repository, not of a session, so the required preceding `restoreFromBackup:` could have been executed in some preceding session.

If GemStone was using partial-logging mode at the time restored backup file(s) were written then `restoreFromLog:` is not allowed.

You must be the only user logged in, otherwise an error is generated.

This method requires the FileControl privilege. It is recommended that you run as either DataCurator or SystemUser. This method puts the session into manualBegin transaction mode.

`restoreStatus`

Returns a String describing the current restore status of the Repository, including the next transaction log file or backup file required to continue the restore.

Restore status is an attribute of the Repository, not of the session, and persists across logout/login and stopstone/startstone.

This method requires the FileControl privilege.

`restoreStatusNextFileId`

Returns a SmallInteger, the fileId of the next tranlog or backup that should be restored, or nil if restore not active.

Restore status is an attribute of the Repository, not of the session, and persists across logout/login and stopstone/startstone.

This method requires the FileControl privilege.

`setArchiveLogDirectories:` *arrayOfDirectorySpecs*
`tranlogPrefix:` *tranlogPrefixString*
`replicateDirectories:` *arrayOfReplicateDirSpecs*
`replicatePrefix:` *replicPrefixString*

Specifies the directories and raw partitions to be searched by subsequent invocation(s) of `restoreFromArchiveLogs`.

The argument *arrayOfDirectorySpecs* must be an Array of one or more Strings. Each String must name a file system directory or raw device specification. An error is generated if any of the directories or devices does not exist. It is not an error if they exist but do not yet contain any tranlogs.

The argument *tranlogPrefixString* must be a String, the file prefix to be used when searching for tranlogs in file systems specified in *arrayOfDirectorySpecs*. The argument may be nil, in which case the value for STN_TRAN_LOG_PREFIX in the stone's configuration file is used.

The argument *arrayOfReplicateDirSpecs* must be nil, or an Array of zero or more Strings that specify file system directories and/or raw devices to search for replicate tranlogs that have the file prefix *replicPrefixString*. An error is generated if any of the directories or devices does not exist. It is not an error if they exist but do not yet contain any tranlogs.

The argument *replicPrefixString* must be a String, the file prefix to be used when searching for tranlogs in file systems specified in *arrayOfReplicateDirSpecs*. The argument may be nil, in which case the value for STN_REPL_TRAN_LOG_PREFIX in the stone's configuration file is used.

Does not require privileges. Does not require that you be the only user logged in. However, a subsequent restore operation to use the state set by this method will require that the Repository be in restore state, and that you have FileControl privilege and that you then be the only user logged in.

`timeToRestoreTo:` *aDateTime*

Sets the time at which `restoreFromLog:` and `restoreFromCurrentLogs` will stop. The restore will stop at the first checkpoint which originally occurred at or after *aDateTime*. If `timeToRestoreTo:` has not been used since `restoreFromBackup:` completed, then restores will proceed to the end of the specified transaction log(s).

An error is generated if *aDateTime* preceeds the time of the last restored checkpoint, as shown by `restoreStatus`. An error is generated if the receiver is not in restore-from-log state.

Execution of `restoreFromBackup:` or `commitRestore` will cancel the effect of any previous execution of `timeToRestoreTo:`.

If restore has stopped at a time specified by this method, then a subsequent `restoreFromLog:` or `restoreFromCurrentLogs` may be used to continue restoring past the time specified by the last `timeToRestoreTo:`. Alternatively, `timeToRestoreTo:` can be used to specify another point in time before continuing the restore.

This method requires the FileControl privilege.

Backward Compatibility

Methods in this category are obsolete and are provided only for compatibility with earlier releases of GemStone. They will be removed in a future release.

`scavengePagesWithPercentFree:` *aPercent*

Obsolete in GemStone 5.0. The GcGem automatically reclaims pages with greater than 10 percent free in an operation that does not cause concurrency conflicts.

Class Management

`listInstances:` *anArray*

Returns a list of instances on the receiver that belong to one of the classes listed in *anArray*. The result of this method is an Array of Sets, where the contents of each set consists of all instances whose class is equal to the corresponding element in *anArray*.

Warning:
You may retrieve instances to which you have no read access, so this method is mostly of use to SystemUser.

If *anArray* contains multiple occurrences of a class, then the result will contain corresponding multiple occurrences of the same Set that lists the instances of that class.

If *anArray* contains an element that is not a kind of Behavior, an error is generated.

Scans the entire Repository at least once.

If the argument *anArray* contains more than 2000 unique elements then the entire Repository will be scanned once for each group of 2000 unique elements, or fraction thereof.

`listReferences:` *anArray*

Returns a list of instances in the receiver that have a reference to one of the objects specified in *anArray*. The result of this method is an Array of Sets, where the contents of each set consists of all instances that have a reference to the corresponding element in *anArray*. Instances to which you have no read authorization are silently omitted.

The result contains both permanent and temporary objects. The temporary objects found may vary from run to run.

Warning:
This method is very expensive. It scans the entire Repository and looks at every instance variable of every object.

`listReferences:` *anArray* `withLimit:` *aSmallInt*

Returns a list of instances in the receiver that have a reference to one of the objects specified in *anArray*. The result of this method is an Array of Sets, where the contents of each set consists of instances that have a reference to the corresponding element in *anArray*. The number in each set is limited by *aSmallInt*. If an instance for which you have no read authorization is found, the result set contains nil.

The result contains both permanent and temporary objects. The temporary objects found may vary from run to run.

Warning:
This method is very expensive. It scans the entire Repository and looks at every instance variable of every object.

Clustering

`extentForPage:` *aPageId*

Returns a SmallInteger specifying an offset into the result from the `fileNames` method.

The argument *aPageId* is an Integer, such as the result from the `Object | page` method, specifying a page in the receiver.

Copying

`copy` Disallowed.

Extent Operations

`createExtent:` *extentFilename*

Creates a new Extent with a file named *extentFilename* (a String). The new Extent has no maximum size.

This method updates the DBF_EXTENT_NAMES stone option. It does not require the system to be in single-user mode.

If the given file already exists, then this method generates an error and the given Extent is not added to the logical Repository.

Creating an extent with this method bypasses any setting the user may have specified for the DBF_PRE_GROW option at system startup. As extents created with this method have no maximum size, they cannot be pre-grown.

If GemStone is being run using weighted disk resource allocation, then the new Extent will be given a weight equal to the average weight of all other extents.

This method requires the FileControl privilege. It is recommended that you run as either DataCurator or SystemUser.

`createExtent:` *extentFilename* `withMaxSize:` *aSmallInteger*

Creates a new Extent with the given *extentFilename* (aString) and sets the maximum size of that Extent to the the given size.

The size must be a non-zero positive integer representing the maximum physical size of the file in megabytes.

This method updates the DBF_EXTENT_NAMES and DBF_EXTENT_SIZES stone options. It does not require the system to be in single-user mode.

Note:
The extent is created with the default ownership and permissions of the stone process. If this is not the same as the ownership and permissions of the other extents or replicates of extents, then Unix utilities must be used to change the ownership or permissions of the new file; such changes may be made without stopping the stone, and should be made a soon as possible, to avoid other sessions encountering authorization errors.

If the given file already exists, then this method generates an error and the given Extent is not added to the logical Repository.

If the stone option DBF_PRE_GROW is set to true, then this method will cause the newly created extent to be pre-grown to the given size. If the grow fails, then this method returns an error and the new Extent is not added to the logical Repository.

If GemStone is being run using weighted disk resource allocation, then the new Extent will be given a weight equal to the average weight of all other Extents.

This method requires the FileControl privilege. It is recommended that you run as either DataCurator or SystemUser.

`numberOfExtents` Returns the number of active extents.

`shrinkExtents`

Truncate all Extents of the Repository to remove internal free space between the last used page in each extent and the end of the file containing the extent. Has no effect for extents on a raw disk partition.

This may be run while the system is in normal operation and does not commit the current transaction. System performance may be degraded while this operation is in progress.

This method requires the FileControl privilege. It is recommended that you run as either DataCurator or SystemUser.

If DBF_PRE_GROW is enabled in the configuration file then this the extents will be grown again the next time stone is restarted, thus cancelling the effect of this method.

The Garbage Collector session is shut down for the duration of this method.

`validateExtentId:` *anExtentId*

Returns the argument if it is valid, otherwise generates an error.

Formatting

`printOn:` *aStream*

Puts a displayable representation of the receiver on the given stream.

Garbage Collection

`addGcCandidates:` *anArray*

Adds an array of candidates to the global queue of GcCandidates. When the transaction that this method is executed in is committed, the candidates will be visible to a session that performs the `markGcCandidates` method.

`auditWithLimit:` *sizeLimit*

Checks all objects in GemStone for consistency. (Compare with Repository's instance method `repairWithLimit:`.) A description of errors found is written to standard output, along with statistics about the Repository. The statistics report will not include any objects smaller than the specified *sizeLimit* (number of bytes or OOPs).

This method should be executed from topaz -l (the linked version of Topaz).

This method aborts the current transaction.

This method requires the GarbageCollection privilege.

The Garbage Collector session is shut down for the duration of this method.

If this session is the only session logged, logins are disabled for the duration of this method, scavenging is forced to complete, and additional consistency checks are made during the audit. If other users were found to be logged in, then scavenging is not completed, logins are not disabled, and the audit performs less checking of the Repository.

A GciHardBreak during this method terminates the session.

This method raises the special error #3021. Topaz performs special processing on this result to determine whether or not the audit succeeded. The Topaz 'expectvalue true' command matches the result of a successful object audit.

`findDisconnectedObjects`

This method helps determine the kinds of objects that are dead in GemStone and thus allows you to tune your code to help prevent unnecessary objects from being stored on disk.

This method returns an Array containing a list of (non-private) objects that are not transitively connected to any permanent object, based on the GemStone state as viewed by the transaction in which this method is executed. These objects are considered possibly dead and could be reclaimed by execution of a `markForCollection`.

It is important that the application disconnect the returned Array after examining it to avoid having these possible dead objects be inadvertantly connected to GemStone permanently.

This method requires the GarbageCollection privilege.

`markForCollection` Performs a garbage collection analysis of all permanent objects on disk. Every object in the receiver that cannot be reached from AllUsers is marked for subsequent reclaiming of storage space.

This method aborts the current transaction, empties the GcCandidates queue and commits, runs the analysis while outside of a transaction and then reenters a transaction if the session was in a transaction at the start of this method.

When this method completes successfully, it generates an error message of the following form:

Successful completion of Garbage Collection
found *anInt* live objects,
found *anInt* dead objects,
occupying *anInt* bytes

After this method completes, the GcGem process automatically reclaims the space occupied by the dead objects. Space is not reclaimed until other sessions have either committed or aborted the transactions that were concurrent with this method.

This method may fail with an error if epoch garbage collection or `markGcCandidates` is in progress at the time it is executed. In this case, `markForCollection` should be retried after waiting a few minutes.

This method requires the GarbageCollection privilege. A GciHardBreak during this method terminates the session.

`markGcCandidates`

Performs a garbage collection analysis on the objects in GcCandidates by scanning the Repository for other references to them. Objects are marked for subsequent reclaiming of storage space if the only references are from other candidates.

Use the `Repository | addGcCandidates:` method to add elements to the queue.

This method may fail with an error if epoch garbage collection or `markForCollection` is in progress at the time it is executed. In this case, `markGcCandidates` should be retried after waiting a few minutes.

This method empties the GcCandidates queue into a temporary Array and commits. It then runs the analysis while outside of a transaction.

When this method completes successfully, it returns an Array containing at least two elements:

Array[1] = number of possible dead Objects found.
Array[2] = number of bytes in the possible dead objects.

If there are entries in the array after the second then:
Array[3..objSize] = the entries in the GcCandidates queue that were not eligible for collection.

Space for the objects determined to be dead is not reclaimed until other sessions have either committed or aborted the transactions that were concurrent with this method.

This method requires the GarbageCollection privilege.

A GciHardBreak during this method will terminate the session.

`objectAudit`

Checks all objects in GemStone for consistency.

This method is equivalent to the message

```
auditWithLimit: 100000.
```

See `auditWithLimit:` for further documentation.

This method should be executed from topaz -l (the linked version of Topaz).

This method aborts the current transaction.

The Garbage Collector session is shut down for the duration of this method.

A GciHardBreak during this method will terminate the session.

This method raises the special error #3021. Topaz performs special processing on this result to determine whether or not the audit succeeded. The Topaz 'expectvalue true' command matches the result of a successful object audit.

`pagesWithPercentFree:` *aPercent*

This method returns an Array containing the following statistics:

1. The total number of data pages processed.
2. The sum in bytes of unused space in all data pages. This quantity is a measure of data fragmentation in the receiver.
3. The number of bytes in a page.
4. The number of data pages that have at least the specified percentage of unused space.

Do not confuse unused space on a page with free space (unused pages) in a Repository or Extent. See the `freeSpace` and `freeSpaceInExtent:` methods for more information.

This method requires the GarbageCollection privilege.

A GciHardBreak during this method will terminate the session.

`reclaimAll`

Explicitly triggers the reclamation of all shadowed and dead objects. Returns true if the reclaim process completes normally; returns false if it is not able to do the reclaim because this is not the only session logged in.

This method requires the GarbageCollection privilege.

`repairWithLimit:` *sizeLimit*

Checks all objects in GemStone for consistency and repairs any errors found. A description of errors found and repaired is written to standard output, along with statistics about the Repository. The statistics report does not include any objects smaller than the specified *sizeLimit* (number of bytes or OOPs).

This method should be executed from topaz -l (the linked version of Topaz).

This method requires the GarbageCollection privilege. In addition, you must be the only user logged into GemStone.

This method aborts the current transaction.

The Garbage Collector session is shut down for the duration of this method.

A GciHardBreak during this method will terminate the session.

The result of this method is error 3021, which Topaz will process specially to determine whether or not the repair found errors. The Topaz command expectvalue true will match the result of a repair which found no errors.

Replicate Operations

`createReplicateOf:` *extentFilename* `named:` *replicateFilename*

Create a replicate for the extent with the given name. (*extentFilename* and *replicateFilename* are both Strings.)

This method requires the FileControl privilege. It is recommended that you run as either DataCurator or SystemUser. In addition, you must be the only user logged into the system.

Note:
The extent is created with the default ownership and permissions of the stone process. If this is not the same as the ownership and permissions of the other extents or replicates of extents, then Unix utilities must be used to change the ownership or permissions of the new file; such changes may be made without stopping the stone.

Each extent is limited to one replicate. If the given extent already has a replicate, this method generates an error.

If the given Extent is not a member of the logical Repository, this method generates an error.

If the given replicate file already exists, then this method generates an error.

If the stone option DBF_PRE_GROW is set to true, then this method will cause the newly created replicate to be pre-grow to the maximum size of the extent it mirrors. If the grow fails, then this method returns an error and the new replicate will not be made available .

The Garbage Collector session is shut down for the duration of this method.

`disposeReplicate:` *replicateFilename*

Removes the given replicate from, whose name is specifies as aString, from the system.

This method requires the FileControl privilege. It is recommended that you run as either DataCurator or SystemUser. In addition, you must be the only user logged into the system.

If the given replicate is not a real Repository replicate, this method generates an error.

The Garbage Collector session is shut down for the duration of this method.

Repository Usage Reporting

`fileSize`
Returns an integer giveing the total physical size, in bytes, of all of the physical extents that compose the logical Repository.

`fileSizeOfExtent:` *extentFilename*
Returns the physical size, in bytes, of the extent with the given name.

If the given file is not an active member of the logical Repository represented by the receiver, then this method generates an error.

`fileSizeReport`
Returns a string which reports on the name, size, and amount of free space for each extent and the size and free space of the entire logical Repository.

`freeSpace`
Returns an integer that gives the number of bytes of free space in the logical Repository. This quantity is equal to the number of free pages in the Repository times the size of a page.

`freeSpaceInExtent:` *extentFilename*
Returns the number of bytes of free space in the extent with the given name. This quantity is equal to the number of free pages in the extent times the size of a page.

If the given file is not an active member of the logical Repository represented by the receiver, then this method generates an error.

`numToMByteString:` *aNumber*
Convert a number representing a file size in bytes to a formatted string reporting the size in megabytes.

`pageSize`
Returns size in bytes of a disk page in the Repository.

Storing and Loading

`writeTo:` *aPassiveObject* Instances of Repository cannot be converted to passive form. This method writes nil to *aPassiveObject* and stops GemStone Smalltalk execution with a notifier.

Transaction Logging

`addTransactionLog:` *deviceOrDirectory* `replicate:` *replicateSpec* `size:` *aSize*

Add *deviceOrDirectory* to the configuration parameter STN_TRAN_LOG_DIRECTORIES. If STN_TRAN_LOG_REPLICATES is not empty, then *replicateSpec* must be a valid device or directory. If STN_TRAN_LOG_REPLICATES is empty (transaction log replicates are not being used), then *replicateSpec* must be an empty String.

The *aSize* argument must be a positive SmallInteger; it is added to the value of the STN_TRAN_LOG_SIZES configuration parameter.

This method requires the FileControl privilege.

`currentLogDirectoryId`

Returns a positive SmallInteger, which is the one-based offset specifying the element of the configuration list STN_TRAN_LOG_DIRECTORIES for the current transaction log. (See also the `currentLogFile` method.)

`currentLogFile` Returns a String containing the file name of the transaction log file to which log records are being appended. If the result is of size 0, then a log has failed and a replicate is being used.

`currentLogFileId` Returns a positive SmallInteger, which is the internal fileId of the current transaction log.

`currentLogReplicate` Returns a String containing the file name of the transaction log replicate file to which log records are being appended. The result is a String of size 0 if the current log is not replicated.

`currentTranlogSizeMB` Returns an Integer that is the size of the currently active transaction log in units of Megabytes.

`logOriginTime`

Returns the log origin time of the receiver. This is the time when a stone started a new sequence of log files for the receiver. A new sequence of logs is started if one of the following occurs:

- Stone is started using extents that were cleanly shutdown, and without any log files being present.
- Stone is started using extents and no pre-existing logs using the 'startstone -N' command.
- The `commitRestore` method is executed, and during preceding restore operations we restored a log file with fileId greater than the fileId of the log file being written to during the restore.

`oldestLogFileIdForRecovery`

Returns a positive SmallInteger, which is the internal fileId of the oldest transaction log needed to recover from the most recent checkpoint, if the stone were to fail right now.

`startNewLog`

Causes the most current transaction log to be closed and a new transaction log to be opened for writing. The location of the new log is controlled by the STN_TRAN_LOG_DIRECTORIES configuration file parameter.

If GemStone is running in partial logging mode, then a preceding transaction log may be deleted. See documentation on the STN_TRAN_FULL_LOGGING configuration parameter for more details.

Returns a SmallInteger, the fileId of the new log.

This method requires the FileControl privilege.

Updating

`at:` *anIndex* `put:` *aValue*	Disallowed.
`atAllPut:` *anObject*	Disallowed.
`name:` *aString*	Redefines the logical **name** of the receiver to be *aString*.
`size:` *anInteger*	Changes the indexed size of the receiver to *anInteger*. May not be used to grow the receiver. Growing a Repository should only be done by sending the following message: `Segment newInRepository: SystemRepository` If *anInteger* is less than the current size of the receiver, the receiver is shrunk accordingly. If *anInteger* is greater than the current size of the receiver, an error is generated.

Class Protocol

Instance Creation

`new`	Disallowed.
`new:` *anInteger*	Disallowed.

ScaledDecimal

ScaledDecimal stores numerical values as a rational number, represented by a numerator and denominator that are Integers. Since the numerator and denominator can be carried to arbitrary precision, ScaledDecimal can represent any rational number without loss of precision. It also calculates based upon fractional arithmetic, and thus produces numerical results without loss of precision.

ScaledDecimal also provides for automatic rounding to a fixed precision after the decimal point when converting to and from other types, such as String.

One useful application of this kind of number is for financial instruments, which are always rounded off, but usually need more digits than a floating number can accurately express in order not to lose precision during computation.

Superclasses Number, Magnitude, Object

Named Instance Variables

numerator — An Integer that represents the numerator of the rational value of the instance.

denominator — A positive Integer that represents the denominator of the rational value of the instance.

scale — A non-negative Integer that represents the number of decimal places of precision to the right of the decimal point.

Instance Format Pointer, Nonindexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`at:` *anIndex* Disallowed.

`at:` *anIndex* `put:` *aNumber*
Disallowed. You may not change the value of a ScaledDecimal.

`denominator` Returns the **denominator** of the receiver.

`instVarAt:` *anIndex* `put:` *aValue*
Disallowed. You may not change the value of a ScaledDecimal.

`numerator`	Returns the **numerator** of the receiver.
`scale`	Returns the **scale** of the receiver.
`size:` *anInteger*	Disallowed. You may not change the size of a ScaledDecimal.

Arithmetic

`*` *aScaledDecimal*	Returns the result of multiplying the receiver by *aScaledDecimal*.
`+` *aScaledDecimal*	Returns the sum of the receiver and *aScaledDecimal*.
`–` *aScaledDecimal*	Returns the difference between the receiver and *aScaledDecimal*.
`/` *aScaledDecimal*	Returns the result of dividing the receiver by *aScaledDecimal*.
`negated`	Returns a Number that is the negation of the receiver.
`reciprocal`	Returns the reciprocal of the receiver.

Comparing

`<` *aScaledDecimal*	Returns true if the receiver is less than *aScaledDecimal*; returns false otherwise.
`=` *aScaledDecimal*	Returns true if the receiver is equal to *aScaledDecimal*; returns false otherwise.

Converting

`asDecimalFloat`	Returns an instance of DecimalFloat that has the value of the receiver.
`asFloat`	Returns an instance of Float that has the value of the receiver.
`asScaledDecimal`	Returns a ScaledDecimal representing the receiver.

Formatting

`asString`	Returns a String of the form ’123.56 for a number with scale = 2.
`withScale:` *newScale*	Returns the receiver with the new **scale**.

Storing and Loading

`writeTo:` *passiveObj* — Converts the receiver to its passive form and writes that information on *passiveObj*.

Testing

`isZero` — Returns true if the receiver is zero.

Truncation and Rounding

`truncated` — Returns the integer that is closest to the receiver, on the same side of the receiver as zero is located.

Updating

`reduced` — Returns a ScaledDecimal determined by finding the greatest common divisor of the **numerator** and **denominator** of the receiver.

Class Protocol

Instance Creation

`fromString:` *aString* — Given *aString* such as '34.23', returns an instance of ScaledDecimal with appropriate numerator and denominator, and with scale equal to the number of digits to the right of the decimal point. Characters in *aString* after the first character which is neither a digit or decimal point are ignored.

`numerator:` *numerator* `denominator:` *denominator* `scale:` *scale*

Returns an instance of ScaledDecimal with the given numerator and denominator. If that ScaledDecimal can be reduced, this method returns the corresponding Integer instead.

If either argument (*numerator* or *denominator*) is not an Integer, that argument is truncated to the corresponding Integer.

Storing and Loading

`loadFrom:` *passiveObj* — Reads from *passiveObj* the passive form of an object. Converts the object to its active form by loading the information into a new instance of the receiver. Returns the new instance.

Segment

Each Repository is composed of an integral number of Segments. A Segment has the following properties:

- Ownership. This is the smallest unit of ownership for accounting and authorization purposes. Each Segment is owned by one and only one user.
- Authorization unit. A user may control access to objects by placing them in a Segment with a known authorization. Prior to reading or writing an object in a Segment, users must be authorized to perform the desired action. However, during a transaction, once a user is authorized to read one object in a Segment, that user can read any object in the same Segment.

Segments and Repositories are discussed in more detail in the *GemStone Programming Guide*.

Superclasses

Object

Class Variables

AuthorizationSymbols — This variable is obsolete in GemStone 4.1 and is provided only for compatibility with earlier releases.

An array of authorization symbols. The authorization for each group, the owner, and the world is encoded in two bits in the authorizations instance variable. Taking these two bits as an integer and adding one gives an index into this array of the corresponding authorization symbol.

Named Instance Variables

itsRepository — The Repository containing the Segment.

itsOwner — A UserProfile indicating the owner of the Segment.

groupsRead — An IdentitySet of canonical Strings. Each String must be an element of AllGroups, and represents a group of users who are authorized to access the Segment for reading.

groupsWrite — An IdentitySet of canonical Strings. Each String must be an element of AllGroups, and represents a group of users who are authorized to access the Segment for writing.

ownerAuthorization — A SmallInteger specifying authorization for the owner to access the Segment. 0 = no access, 1 = read access, 2 = write access.

worldAuthorization — A SmallInteger specifying world authorization to access the Segment. 0 = no access, 1 = read access, 2 = write access.

spare1 — Reserved for future use.

Instance Format Pointer, Nonindexable, Variant

Subclass Creation Disallowed

Instance Protocol

Accessing

`groups` Returns an IdentitySet, the set of user groups (Strings) that are explicitly authorized to read or write in this Segment.

`number` Returns the index of the receiver in its Repository. Return -1 if the Repository does not contain the receiver.

`owner` Returns the UserProfile of the receiver's owner.

`repository` Returns the Repository containing the receiver.

Accessing Authorization

`authorizationForGroup:` *aGroupString*
Returns a Symbol that defines whether the specified group is authorized to write (and read) in this Segment (#write), to read only (#read), or neither (#none).

`authorizationForUser:` *aUserProfile*
Returns a Symbol that describes the authorization that the given user has for the receiver.

`currentUserCanRead` Returns true if the current user has read authorization for the receiver, false if not.

`currentUserCanWrite` Returns true if the current user has write authorization for the receiver, false if not.

`groupsWithAuthorization:` *anAccessSymbol*
Returns an IdentitySet of group Strings of all groups with the authorization of *anAccessSymbol* (one of #write, #read, or #none) for the receiver.

`ownerAuthorization` Returns a Symbol that defines whether the Segment's owner is authorized to write (and read) in this Segment (#write), to read only (#read), or neither (#none).

`usersWithAuthorization:` *anAccessSymbol*
Returns a UserProfileSet containing all users with the authorization of *anAccessSymbol* (one of #write, #read, or #none) for the receiver.

`worldAuthorization` Returns a Symbol that defines whether all system users are authorized to write (and read) in this Segment (#write), to read only (#read), or neither (#none).

Backward Compatibility

Methods in this category are obsolete and are provided only for compatibility with earlier releases of GemStone. They will be removed in a future release.

`groupNo:` *groupIndex* `group:` *aGroupString*
`authorization:` *anAuthorizationSymbol*
Obsolete in GemStone 4.1. Use the `group:authorization:` method instead.

Clustering

`clusterDepthFirst` This method clusters the receiver. (Overrides the inherited method, which also clusters all instance variables.) Returns true if the receiver has already been clustered during the current transaction, false otherwise.

Copying

`copy` Disallowed. To create a new Segment, use newInRepository: instead.

Execution

`setCurrentWhile:` *aBlock*

Sets the receiver to be the current segment while the given block executes. Catches all errors and reinstalls the current segment to avoid having the receiver be left as the current segment. Returns the result of evaluating the zero-argument block.

Note that the argument, *aBlock*, should not have any returns in it. Such returns will bypass the restoration of the current segment that is normally performed after the block has been evaluated.

Formatting

`asString`

Returns a formatted String that contains the following information:

The name of the Repository containing the receiver and the index of the receiver in that Repository.

- The user id of the receiver's owner.
- The Symbol that defines each user group which is authorized to read or write in the receiver.
- Information about whether the receiver's owner, each group, or all users are authorized to read or write in the receiver.

For example, the message

```
System myUserProfile defaultSegment
   asString
```

returns a String that resembles the following:

aSegment, Number 1 in Repository SystemRepository Owner SystemUser write World read

If the receiver is not a Segment in the SystemRepository, the String contains 'NOT IN REPOSITORY' in place of the index.

Storing and Loading

`writeTo:` *aPassiveObject* — Instances of Segment cannot be converted to passive form. This method writes nil to *aPassiveObject* and stops GemStone Smalltalk execution with a notifier.

Updating

`owner:` *aUserProfile* — Redefines the receiver's owner to be the user represented by *aUserProfile*. To execute this method, you must be authorized to write in the receiver's Segment (customarily, the DataCurator Segment).

Updating Authorization

In general, if you are not the data curator, you may only change the authorization for a Segment if you are the owner of that Segment. To execute an authorization message on a Segment you do not own, you must have the SegmentProtection privilege.

Exercise caution when changing the authorization for any Segment that a user may be using as his or her default or current Segment (whether or not the user owns the affected Segment). If a user attempts to commit a transaction, but has created objects in a Segment for which he or she no longer has write authorization, an error will be generated. In addition, if a user is no longer authorized to write in his or her default Segment, the user's GemStone session will be terminated and the user will be unable to log back in to GemStone.

`group:` *aGroupString* `authorization:` *anAuthorizationSymbol*

Redefines the authorization for the specified group to one of the following authorization Symbols:

- #write (members of the group can both read and write in this Segment).
- #read (read only).
- #none (neither read nor write permission).

This method generates an error if *aGroupString* is not an element of the global object AllGroups, or if *anAuthorizationSymbol* is not one of (#read, #write, #none).

Requires the SegmentProtection privilege, if the segment is not owned by the GemStone UserProfile under which this session is running.

`ownerAuthorization:` *anAuthorizationSymbol*

Redefines the authorization for the Segment's owner to one of the following authorization Symbols:

- #write (the Segment's owner can both read and write in this Segment).
- #read (read only).
- #none (neither read nor write permission).

Generates an error if *anAuthorizationSymbol* is not one of (#read, #write, #none).

Requires the SegmentProtection privilege, if the segment is not owned by the GemStone UserProfile under which this session is running.

`worldAuthorization:` *anAuthorizationSymbol*

Redefines the authorization for all users to one of the following authorization Symbols:

- #write (all users can both read and write in this Segment).
- #read (read only).
- #none (neither read nor write permission).

Generates an error if *anAuthorizationSymbol* is not one of (#read, #write, #none).

Requires the SegmentProtection privilege, if the segment is not owned by the GemStone UserProfile under which this session is running.

Class Protocol

Instance Creation

`new`

Disallowed. To create a new Segment, use `newInRepository:` instead.

`newInRepository:` *aRepository*

Returns a new Segment in the Repository *aRepository*. If the maximum number of Segments has already been created for *aRepository*, this generates an error. The new Segment has the default authorization of W----- (owner can read and write).

Requires the SegmentCreation privilege.

SelectBlock

SelectBlock is a concrete subclass of BlockClosure that supports selection blocks for associative access.

Superclasses BlockClosure, Object

Named Instance Variables

queryBlock — An ExecutableBlock. The firstPC instance variable of this block refers to the first bytecode to execute to begin an associative access query, and the lastPC instance variable refers to the last bytecode of the associative access query.

iterationBlock — An ExecutableBlock. The firstPC instance variable of this block refers to the first bytecode to execute to begin execution of this block in GemStone Smalltalk (no associative access), and the lastPC instance variable refers to the last bytecode of the block's GemStone Smalltalk execution.

Instance Format Pointer, Nonindexable, Variant

Subclass Creation Disallowed

Instance Protocol

Accessing

`iterationBlock` Returns the value of the **iterationBlock** instance variable.

`queryBlock` Returns the value of the **queryBlock** instance variable.

Block Evaluation

`value:` *anObject* Returns the value of the receiver evaluated with *anObject* as its argument. A SelectBlock can only take one argument.

SequenceableCollection

SequenceableCollection is an abstract superclass for collections that define a consistent ordering on their elements. You can think of the elements as forming a sequence of objects, numbered from 1 to n. You can use the Integer i as an index to refer to the ith element in that sequence. The elements are said to be indexable.

The indexability of SequenceableCollections should not be confused with indexes that are specially built on UnorderedCollections to improve their performance when they search their elements (using associative access).

Superclasses Collection, Object

Named Instance Variables None

Instance Format Pointer, Indexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`after:` *anObject* Returns the object immediately following the first object which is equal to *anObject* in the receiver.

`before:` *anObject* Returns the object immediately before the first object which is equal to *anObject* in the receiver.

`first` Returns the first element of the receiver. Generates an error if the receiver is empty.

`last` Returns the last element of the receiver. Generates an error if the receiver is empty.

Adding

Methods in the Adding category modify their receivers. They generally execute faster than methods in the Concatenating category, which do not modify their receivers. Please see that category for more comparative information.

`add:` *newObject*
Makes *newObject* one of the receiver's elements and returns *newObject*. The new element is actually added as the last element of the receiver. This method is therefore equivalent to `addLast:`.

`add:` *newObject* `after:` *target*
Adds *newObject* to the receiver immediately after the first element that is equal to *target*. An object immediately follows another if its index is one greater than that of the other. Returns *newObject* if the operation is successful. Raises an error if the operation fails.

`add:` *newObject* `before:` *target*
Adds *newObject* to the receiver immediately before the first element that is equal to *target*. An object immediately precedes another if its index is one less than that of the other. Returns *newObject* if the operation is successful. Raises an error if the operation fails.

`addAll:` *aCollection* `afterIndex:` *index*
Adds all the elements of *aCollection* to the receiver in the traversal order defined by the do: method for *aCollection*. Inserts the new elements into the receiver immediately after the element in the receiver at position *index*. If *index* is equal to zero, inserts the elements of *aCollection* at the beginning of the receiver. Returns *aCollection*.

The argument *index* must be a non-negative integer less than or equal to the receiver's size.

`addAll:` *aCollection* `before:` *target*
Adds all the elements of *aCollection* to the receiver immediately before the first element that is equal to *target*. An object immediately precedes another if its index is one less than that of the other. Returns *aCollection* if the operation is successful. Raises an error if the operation fails.

`addAll:` *aCollection* `beforeIndex:` *index*

Adds all the elements of *aCollection* to the receiver in the traversal order defined by the do: method for *aCollection*. Inserts the new elements into the receiver immediately before the element in the receiver at position *index*. If *index* is equal to the receiver's size plus one, inserts the elements of *aCollection* at the end of the receiver. Returns *aCollection*.

The argument *index* must be a positive integer less than or equal to the receiver's size plus one.

`addLast:` *newObject* Adds *newObject* as the last element of the receiver and returns *newObject*.

`insertAll:` *aCollection* `at:` *anIndex*

Inserts all the elements of *aCollection* into the receiver beginning at index *anIndex*. Returns *aCollection*.

The argument *anIndex* must be greater than or equal to one. If *anIndex* is one greater than the size of the receiver, appends *aCollection* to the receiver. If *anIndex* is more than one greater than the size of the receiver, generates an error.

`insertObject:` *anObject* `at:` *anIndex*

Inserts *anObject* into the receiver at index *anIndex* and returns *anObject*.

Backward Compatibility

Methods in this category are obsolete and are provided only for compatibility with earlier releases of GemStone. They will be removed in a future release.

`+` *aSequenceableCollection* Obsolete in GemStone 5.0. Use the `,` method instead.

`deleteFrom:` *startIndex* `to:` *stopIndex*

Obsolete in GemStone 5.0. Use the `removeFrom:to:` method instead.

`deleteObjectAt:` *anIndex*

Obsolete in GemStone 5.0. Use the `removeAtIndex:` method instead.

`eq:` *aSequenceableCollection*

Obsolete in GemStone 5.0. Use the = method instead.

`indexOfValue:` *anObject* Obsolete in GemStone 5.0. Use the `indexOf:` method instead.

`insert:` *aSequenceableCollection* `at:` *anIndex*
Obsolete in GemStone 5.0. Use the `insertAll:at:` method instead.

`removeAll:` *aCollection* `ifAbsent:` *errorBlock*
Obsolete in GemStone 5.0.

`removeIndex:` *index* Obsolete in GemStone 5.0. Use the `removeAtIndex:` method instead.

`removeValue:` *oldObject* Obsolete in GemStone 5.0. Use the remove: method instead.

Comparing

`=` *aSequenceableCollection* Returns true if all of the following conditions are true:

1. The receiver and *aSequenceableCollection* are of the same class.
2. The two collections are the same size.
3. The corresponding elements of the receiver and *aSequenceableCollection* are equal.

`at:` *anIndex* `equals:` *aSequenceableCollection*
Returns true if *aSequenceableCollection* is contained in the receiver starting at index *anIndex*. Returns false otherwise.

`hash` Returns a numeric hash key for the receiver.

`hasIdenticalContents:` *aSequenceableCollection*
Returns true if all of the following conditions are true:

1. The receiver and *aSequenceableCollection* are of the same class.
2. The two collections are the same size.
3. The corresponding elements of the receiver and *aSequenceableCollection* are identical.

Concatenating

Methods in the Concatenating category do not modify their receivers. They copy their receivers and then apply the concatenation to the copy. Thus, they generally execute slower than methods in the Adding category, which do modify their receivers.

Consider the following code example involving Strings:

```
| n result |
n := 1000.
result := String new.
n timesRepeat: [result := result , $x.].
^ result
```

Each time through the loop, this code first generates a new instance of String, a copy of the previous result, to which it then adds one character. In n times through the loop then, n * (n - 1) / 2 characters are copied, and n characters are added. Thus, the time complexity to execute such a loop is proportional to n * (n + 1) / 2. Space and garbage collection overhead can be expensive, too.

The following code example executes in time that is proportional to n, without any of the space and garbage collection overhead of the previous example:

```
| n result |
n := 1000.
result := String new.
n timesRepeat: [ result add: $x].
^ result
```

The result is the same in both examples.

, *aSequenceableCollection* Returns a new instance of the receiver's class that contains the elements of the receiver followed by the elements of *aSequenceableCollection*.

Warning:
Creating a new instance and copying the receiver take time. If you can safely modify the receiver, it can be much faster to use the `addAll:` *method. See the documentation of the Concatenating category for more details.*

Copying

`copyEmpty` Returns an empty copy of the receiver.

`copyFrom:` *startIndex* `to:` *stopIndex*

Returns a new SequenceableCollection containing the elements of the receiver between *startIndex* and *stopIndex*, inclusive. The result is of the same class as the receiver.

Both *startIndex* and *stopIndex* must be positive integers not larger than the size of the receiver, with *startIndex* <= *stopIndex*.

`copyFrom:` *index1* `to:` *index2* `into:` *aSeqCollection* `startingAt:` *destIndex*

Copies the elements of the receiver between *index1* and *index2*, inclusive, into *aSeqCollection* starting at *destIndex*, overwriting the previous contents. If *aSeqCollection* is the same object as the receiver, the source and destination blocks may overlap. Returns the receiver.

`copyReplaceAll:` *oldSubCollection* `with:` *newSubCollection*

Returns a modified copy of the receiver in which all sequences of elements within the receiver that match the elements of *oldSubCollection* are replaced by the elements of *newSubCollection*.

`copyReplaceFrom:` *startIndex* `to:` *stopIndex* `with:` *aSequenceableCollection*

Returns a copy of the receiver in which all elements in the receiver between indexes *startIndex* and *stopIndex* inclusive have been replaced by those contained in *aSequenceableCollection*.

`copyReplaceFrom:` *startIndex* `to:` *stopIndex* `withObject:` *anObject*

Returns a copy of the receiver in which all elements in the receiver between indexes *startIndex* and *stopIndex* inclusive have been replaced by *anObject*.

`copyReplacing:` *oldObject* `with:` *newObject*

Returns a copy of the receiver in which all occurrences of objects equal to *oldObject* have been replaced by *newObject*.

`copyWith:` *anObject* Returns a copy of the receiver with *anObject* appended at the end.

`copyWithout:` *anObject* Returns a copy of the receiver that does not contain the given object. Comparisons are by equality.

`reverse` Returns a copy of the receiver with its elements in reverse order.

Enumerating

`doWithIndex:` *aBlock* Evaluates the two argument block *aBlock* using each element of the receiver as the first argument and the corresponding index as the second argument. Returns the receiver.

`from:` *startIndex* `to:` *stopIndex* `do:` *aBlock*
Evaluates the one argument block *aBlock* using each element of the receiver starting at *startIndex* and ending at *stopIndex*. Returns the receiver.

`from:` *startIndex* `to:` *stopIndex* `doWithIndex:` *aBlock*
Evaluates the two argument block *aBlock* using each element of the receiver starting at *startIndex* and ending at *stopIndex* as the first argument and the corresponding index as the second argument. Returns the receiver.

`reverseDo:` *aBlock* Evaluates the one-argument block *aBlock* using each element of the receiver, in reverse order, as the argument.

`with:` *aSequenceableCollection* `do:` *aBlock*
Evaluate a two argument block *aBlock* using each element of the receiver as the first argument and the corresponding element of *aSequenceableCollection* as the second argument. Returns the receiver.

Removing

`removeAllSuchThat:` *aBlock*

Removes all elements of the receiver for which *aBlock* returns true. Returns the receiver.

`removeAtIndex:` *anIndex*

Removes the element at the given index. Returns the removed element.

`removeFirst` Removes the first element of the receiver. Returns the removed element.

`removeFrom:` *startIndex* `to:` *stopIndex*

Removes the elements of the receiver from *startIndex* to *stopIndex* inclusive. Returns the receiver.

The size of the receiver is decreased by *stopIndex* - *startIndex* + 1.

`removeLast` Removes the last element of the receiver. Returns the removed element.

Searching

`findFirst:` *aBlock* Returns the index of the first element in the receiver which causes the one argument block, *aBlock*, to evaluate true. Returns 0 if no element in the receiver causes the block to evaluate true.

`findLast:` *aBlock* Returns the index of the last element in the receiver which causes the one argument block, *aBlock*, to evaluate true. Returns 0 if no element in the receiver causes the block to evaluate true.

`indexOf:` *anObject* Returns the index of the first element in the receiver that is equivalent to the argument. If the receiver does not have any elements equivalent to the argument, returns, zero.

`indexOf:` *anObject* `ifAbsent:` *anExceptionBlock*

Returns the index of the first element in the receiver that is equivalent to the argument. If the receiver does not have any elements equivalent to the argument, returns the value of evaluating *anExceptionBlock*.

`indexOf:` *anObject* `startingAt:` *index*

Returns the index of the first element in the receiver that is equivalent to the argument. If the receiver does not have any elements equivalent to the argument, returns 0.

`indexOf:` *anObject* `startingAt:` *index* `ifAbsent:` *anExceptionBlock*

Returns the index of the first element in the receiver that is equivalent to the argument. If the receiver does not have any elements equivalent to the argument, returns the value of evaluating *anExceptionBlock.*

`indexOfSubCollection:` *aSubColl* `startingAt:` *anIndex*

Returns the index of the first element of the receiver where that element and the subsequent ones are equal to those in *aSubColl*. The search is begun in the receiver at starting at *anIndex*. Returns 0 if no match is found.

`indexOfSubCollection:` *aSubColl* `startingAt:` *anIndex* `ifAbsent:` *anExceptionBlock*

Returns the index of the first element of the receiver where that element and the subsequent ones are equal to those in *aSubColl*. The search is begun in the receiver at starting at *anIndex*. Returns the value of evaluating *anExceptionBlock* if no match is found.

Streams

`readStream` Returns a ReadStream on the receiver.

Testing Methods

`verifyAreTrue` Returns true if all elements of the receiver are true. Otherwise, returns false.

`verifyElementsIn:` *aSeqColl*

Check for equality in corresponding elements. Returns true if the receiver and the argument are equal.

Updating

`atAll:` *aCollection* `put:` *anObject*
The argument *aCollection* is a collection of Integers that are used as indexes into the receiver. At each element in the receiver for which its index is in *aCollection*, this method replaces the element with the argument *anObject*. Returns *anObject*.

`atAllPut:` *anObject* Assigns *anObject* to each of the receiver's elements and returns *anObject*.

`first:` *anObject* Stores the given object in the first position in the receiver and returns *anObject*.

`last:` *anObject* Stores the given object in the last position in the receiver and returns *anObject*.

`replaceFrom:` *startIndex* `to:` *stopIndex* `with:` *aCollection*
Replaces the elements of the receiver between the indexes *startIndex* and *stopIndex* with the elements of *aCollection*. Returns the receiver.

`replaceFrom:` *startIndex* `to:` *stopIndex* `with:` *aSeqCollection*
`startingAt:` *repIndex*
Replaces the elements of the receiver between the indexes *startIndex* and *stopIndex* inclusive with the elements of *aSeqCollection* starting at *startIndex*. Returns the receiver.

`replaceFrom:` *startIndex* `to:` *stopIndex* `withObject:` *anObject*
Replaces the elements of the receiver between the indexes *startIndex* and *stopIndex* with *anObject*. Returns the receiver.

Set

A Set is an UnorderedCollection in which any distinct object can occur only once. Adding the same (identical) object to a Set multiple times is redundant. The result is the same as adding it once.

Since a Set is an equality-based collection, different (non-identical) but equivalent (equal) objects are not treated as distinct from each other. In IdentitySets, they are distinct. Adding multiple equivalent objects to a Set yields a Set with the object that was added last. In short, two different elements of a Set are neither identical nor equivalent.

You can create subclasses of Set to restrict the kind of elements it contains. When creating a subclass of Set, you must specify a class as the aConstraint argument. This class is called the element kind of the new subclass. For each instance of the new subclass, the class of each element must be of the element kind.

Superclasses UnorderedCollection, Collection, Object

Named Instance Variables **dict —** A KeyValueDictionary that organizes the elements and element counts for the Set.

Instance Format Nsc, Nonindexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`at:` *anIndex* — Disallowed.

Adding

`add:` *newObject* — Makes *newObject* one of the receiver's elements and returns *newObject*. If an equivalent element is already present in the receiver, the receiver is not modified. A set can have only one occurrence of equivalent objects.

Removing

`removeAll:` *aCollection* — Removes each element of *aCollection* from the receiver and returns the receiver. Generates an error if any element of *aCollection* is not present in the receiver.

Searching

`includesIdentical:` *anObject*
: Returns true if *anObject* is identical to one of the elements of the receiver. Returns false otherwise.

`occurrencesOf:` *anObject*
: In a Set, an object occurs only once if present.

Updating

`at:` *anIndex* `put:` *anObject*
: Disallowed.

`changeToSegment:` *segment*
: Assigns the receiver and its private objects to the given segment.

Class Protocol

Instance Creation

`new`
: Returns an instance of the receiver whose contents are empty.

`new:` *initialSize*
: Returns an instance of the receiver whose contents are empty.

SimpleBlock

A SimpleBlock is an ExecutableBlock that does not refer to any enclosing scope variables. Thus, unlike complex blocks, it needs no variable context at any time.

The GemStone Smalltalk compiler creates all simple blocks.

Superclasses ExecutableBlock, BlockClosure, Object

Named Instance Variables None

Instance Format Pointer, Nonindexable, Variant

Subclass Creation Disallowed

Instance Protocol

Block Evaluation

`value` Return the value of the receiver evaluated with no arguments. If the block expects any arguments, an error is generated.

`value:` *anObject* Return the value of the receiver evaluated with *anObject* as its argument. If the block expects a different number of arguments, an error is generated.

`value:` *firstObject* `value:` *secondObject*
Return the value of the receiver evaluated with the two objects as its arguments. If the block expects a different number of arguments, an error is generated.

`value:` *firstObject* `value:` *secondObject* `value:` *thirdObject*
Return the value of the receiver evaluated with the three objects as its arguments. If the block expects a different number of arguments, an error is generated.

`value:` *first* `value:` *second* `value:` *third* `value:` *fourth*
Return the value of the receiver evaluated with the four objects as its arguments. If the block expects a different number of arguments, an error is generated.

`value:` *first* `value:` *second* `value:` *third* `value:` *fourth* `value:` *fifth*
Return the value of the receiver evaluated with the five objects as its arguments. If the block expects a different number of arguments, an error is generated.

`valueWithArguments:` *argList*
Return the value of the receiver evaluated with the elements of the Array *argList* as arguments. If the block expects a different number of arguments, an error is generated.

Testing

`isSimple` Return true. The receiver is a simple block.

SmallFloat

This class represents 4 byte binary floating point numbers, as defined in IEEE standard 754.

You may not create subclasses of SmallFloat.

Superclasses BinaryFloat, Number, Magnitude, Object

Named Instance Variables None

Instance Format Byte, Indexable, Variant

Subclass Creation Disallowed

Instance Protocol

Accessing

`asFraction` Returns a Fraction that represents the receiver. If the receiver is a NaN, or Infinity, returns the receiver.

`denominator` Returns the denominator of a Fraction representing the receiver.

`numerator` Returns the numerator of a Fraction representing the receiver.

`sign` Returns 1 if the receiver is greater than zero, -1 if the receiver is less than zero, and zero if the receiver is zero.

Arithmetic

`*` *aNumber* Multiply the receiver by *aNumber* and returns the result.

`+` *aNumber* Returns the sum of the receiver and *aNumber*.

`–` *aNumber* Returns the difference between the receiver and *aNumber*.

`/` *aNumber* Divide the receiver by *aNumber* and returns the result.

`negated` Returns a Number that is the negation of the receiver.

Comparing

< *aNumber*	Returns true if the receiver is less than *aNumber*; returns false otherwise.
<= *aNumber*	Returns true if the receiver is less than or equal to a *aNumber*; returns false otherwise.
= *aNumber*	Returns true if the receiver is equal to *aNumber*; returns false otherwise.
~= *aNumber*	Returns true if the receiver is not equal to *aNumber*; returns false otherwise.

Converting

`asDecimalFloat`	Returns a DecimalFloat representing the receiver.
`asFloat`	Returns a Float with the same value as the receiver.
`asSmallFloat`	Returns the receiver.

Formatting

`asString`	Returns a String corresponding to the value of the receiver. Where applicable, returns one of the following Strings: PlusInfinity, MinusInfinity, PlusQuietNaN, MinusQuietNaN, PlusSignalingNaN, or MinusSignalingNaN.

`asStringUsingFormat:` *anArray*

Returns a String corresponding to the receiver, using the format specified by *anArray*. The Array contains three elements: two Integers and a Boolean. Generates an error if any element of the Array is missing or is of the wrong class.

The first element of the Array (an Integer between -1000 and 1000) specifies a minimum number of characters in the result String (that is, the width of the string). If this element is positive, the resulting String is padded with blanks to the right of the receiver. If this element is negative, the blanks are added to the left of the receiver. If the value of this element is not large enough to completely represent the Float, a longer String will be generated.

The second element of the Array (a positive Integer less than 1000) specifies the maximum number of digits to display to the right of the decimal point. If the value of this element exceeds the number of digits required to completely specify the Float, only the required number of digits are actually displayed. If the value of this element is insufficient to completely specify the Float, the value of the Float is rounded up or down before it is displayed.

The third element of the Array (a Boolean) indicates whether or not to display the magnitude using exponential notation. (The value true indicates exponential notation and false indicates decimal notation.)

For example, the number 12.3456 displayed with two different format arrays would appear as follows:

Format	Output
`#(10 5 true)`	`' 1.23456E1'`
`#(10 2 false)`	`'12.34'`

Truncation and Rounding

`truncated`

Returns the integer that is closest to the receiver, on the same side of the receiver as zero is located. In particular, returns the receiver if the receiver is an integer.

Class Protocol

Instance Creation

`fromString:` *aString*

Returns an instance of the receiver, constructed from *aString*. The String must contain only characters representing the object to be created, although leading and trailing blanks are permitted.

If the string represents an exceptional float, it must contain one of the following strings, with leading and trailing blanks permitted: PlusInfinity, MinusInfinity, PlusQuietNaN, MinusQuietNaN, PlusSignalingNaN, or MinusSignalingNaN.

If the string does not conform to the above rules, this method generates an error or returns a SignalingNaN.

If the string is larger than 8191 bytes, an error is generated.

Storing and Loading

`loadFrom:` *passiveObj*

Reads from *passiveObj* the passive form of an object. Converts the object to its active form by loading the information into a new instance of the receiver. Returns the new instance.

SmallInteger

Instances of SmallInteger are an optimization for commonly occurring integers (between -(2 to the 29 power) and (2 to the 29 power - 1)). You may not create subclasses of class SmallInteger. You may not create any new SmallIntegers. Note that all instances of a given SmallInteger refer to a single, unique GemStone object. That is, they are all both equal (=) and identical (==).

Superclasses Integer, Number, Magnitude, Object

Named Instance Variables None

Instance Format Special, Nonindexable, Invariant

Subclass Creation Disallowed

Instance Protocol

Arithmetic

`/` *aNumber*
Returns the result of dividing the receiver by *aNumber*.

`//` *aNumber*
Divides the receiver by *aNumber*. Returns the integer quotient, with truncation toward negative infinity. For example,

```
9//4 = 2
-9//4 = -3
```

The selector `\\` returns the remainder from this division.

`quo:` *aNumber*
Divides the receiver by *aNumber*. Returns the integer quotient, with truncation toward zero. For example,

```
-9 quo: 4 = -2
```

The selector `rem:` returns the remainder from this division.

`\\` *aNumber*
Returns the modulus defined in terms of `//`. Returns a Number with the same sign as the argument *aNumber*. For example,

```
9\\4 = 1
-9\\4 = 3
9\\-4 = -3
```

Bit Manipulation

`asBitString`	Returns a string of 1 and 0 characters representing the bits in the receiver.
`bitAnd:` *aNumber*	Returns an Integer whose bits are the logical and of the receiver's bits and the bits of *aNumber*.
`bitOr:` *aNumber*	Returns an Integer whose bits are the logical or of the receiver's bits and the bits of *aNumber*.
`bitShift:` *shift*	Returns an Integer whose value (in two's-complement representation) is the receiver's value (also in two's-complement representation) shifted left by *shift* bits. Negative arguments *shift* right. Zeros are shifted in from the right in left shifts. The sign bit is extended in right shifts.
`bitXor:` *aNumber*	Returns an Integer whose bits are the logical xor of the receiver's bits and the bits of *aNumber*.
`highBit`	Returns the index of the high-order bit that is set in the bianary representation of the receiver. (If the receiver is negative, takes its absolute value first.) If the receiver is zero, this returns nil.

Clustering

`clusterDepthFirst`	Returns true. (Because SmallIntegers are self-defining objects, this method has no effect.)

Comparing

< *aNumber*	(Optimized selector.) Returns true if the receiver is less than *aNumber*; returns false otherwise. Do not redefine or override this method in this class.
<= *aNumber*	(Optimized selector.) Returns true if the receiver is less than or equal to *aNumber*; returns false otherwise. Do not redefine or override this method in this class.
> *aNumber*	(Optimized selector.) Returns true if the receiver is greater than *aNumber*; returns false otherwise. Do not redefine or override this method in this class.

`hash` Returns a numeric hash index. For a SmallInteger, returns the receiver.

`identityHash` Returns a numeric hash index. For a SmallInteger, returns the receiver.

`~=` *aNumber* (Optimized selector.) Returns true if the receiver is not equal to *aNumber*; returns false otherwise.

Do not redefine or override this method in this class.

Converting

`asDecimalFloat` Returns a DecimalFloat representing the receiver.

`asFloat` Returns a Float representing the receiver.

`asString` Returns a String that indicates the numeric value of the receiver. Positive values do not include a leading + .

Copying

`copy` Overrides the inherited method; you cannot create any new SmallIntegers. Returns the receiver.

Formatting

`printString` Returns a String whose contents are a displayable representation of the receiver.

Class Protocol

Queries

`maximumValue` Returns the maximum allowable SmallInteger value.

`minimumValue` Returns the minimum allowable SmallInteger value.

SortedCollection

A SortedCollection is an OrderedCollection that maintains the order of its elements based on a sort block. In GemStone, SortedCollections are not fixed in length as in other Smalltalk systems.

Superclasses OrderedCollection, SequenceableCollection, Collection, Object

Named Instance Variables **sortBlock —** An ExecutableBlock that defines the sorting criterion. The block must take two arguments, and it should return true if the first argument should precede the second argument, and false if not.

Instance Format Pointer, Indexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`sortBlock` Returns the value of the instance variable **sortBlock**.

Adding

`add:` *anObject* Adds *anObject* to the receiver. Increases the size of the receiver by one. Enforces the sorting order. Returns *anObject*.

`addAll:` *aCollection* Adds the elements of *aCollection* to the receiver. Increases the size of the receiver by the number of elements in *aCollection*. Enforces the sorting order. Returns *aCollection*.

`addLast:` *anObject* Disallowed. Reports an error since SortedCollections have a sorting order that prohibits outside interference.

Copying

`copyFrom:` *startIndex* `to:` *stopIndex*
Installs the receiver's **sortBlock** into the copy.

`copyWithout:` *anObject* Returns a copy of the receiver that does not contain the given object. Comparisons are by equality (not identity).

`insertAll:` *aCollection* `at:` *anIndex*
Disallowed. Reports an error since SortedCollections have a sorting order that prohibits outside interference.

`insertObject:` *anObject* `at:` *anIndex*
Disallowed. Reports an error since SortedCollections have a sorting order that prohibits outside interference.

Searching

`collect:` *aBlock* — Returns an instance of OrderedCollection.

`includes:` *anObject* — Returns true if the argument *anObject* is equal to an element of the receiver. Returns false otherwise.

`includesIdentical:` *anObject*
Returns true if the argument *anObject* is an element of the receiver. Returns false otherwise.

`indexOf:` *anObject* — Returns the index of the first occurrence of *anObject* in the receiver. If the receiver does not contain *anObject*, this returns zero.

`indexOfValue:` *anObject* — Returns the index of the first occurrence of an object equal to *anObject* in the receiver. If the receiver does not contain such an object, this returns zero.

`reject:` *aBlock* — Pass on the sort block.

`select:` *aBlock* — Returns an instance of the receiver's species that has the receiver's sort block.

Storing and Loading

`loadFrom:` *passiveObj* — Reads from *passiveObj* the passive form of an object. Converts the object to its active form by loading the information into the receiver.

Updating

`at:` *index* `put:` *anObject* — Disallowed. Reports an error since SortedCollections have a **sortBlock** that determines the order of their contents.

`atAllPut:` *anObject* — Assigns *anObject* to each of the receiver's elements.

`size:` *anInteger* — If *anInteger* is less than the current size of the receiver, shrinks the receiver, otherwise has no effect.

`sortBlock:` *newBlock* Installs a new sort block in the receiver and forces a resort.

Class Protocol

Instance Creation

`new` Returns a new instance of the receiver with the sort block [:a :b | a <= b].

`new:` *size* Returns a new instance of the receiver with size 0 and the sort block [:a :b | a <= b]. This method is synonymous with new and is provided for compatibility with other Smalltalk dialects that do not have objects that are truly variable in size.

`sortBlock:` *aBlock* Returns a new instance of the receiver with the given sort block.

`sortBlock:` *aBlock* `fromSortResult:` *sortArray*
Returns a new instance of the receiver with the given sort block and contents. The argument *sortArray* is assumed to be in the proper sort order and is installed as the presorted contents of the new instance.

`with:` *aValue* Returns an instance of the receiver containing the argument.

`with:` *aValue* `with:` *val2*
Returns an instance of the receiver containing the arguments.

`with:` *aValue* `with:` *val2* `with:` *val3*
Returns an instance of the receiver containing the arguments.

`withAll:` *aCollection* Returns an instance of the receiver containing the elements of the argument.

`withAll:` *collection* `sortBlock:` *block*
Returns a new instance of the receiver with the given sort block and contents.

Stream

Stream is an abstract superclass that represents the ability to maintain a position reference into a linear sequence of objects. Concrete subclasses are ReadStream and WriteStream.

Superclasses Object

Named Instance Variables None

Instance Format Pointer, Nonindexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`next` (Subclass responsibility, ReadStream only.) Returns the next object that the receiver can access for reading. Generates an error if an attempt is made to read beyond the end of the stream.

Adding

`cr` Adds a newline to the output stream.

`lf` Adds a newline to the output stream.

`nextPut:` *anObject* (Subclass responsibility, WriteStream only.) Inserts *anObject* as the next element that the receiver can access for writing. Returns *anObject*.

`nextPutAll:` *aCollection* Inserts the elements of *aCollection* as the next elements that the receiver can access. Returns *aCollection*. (WriteStream only.)

`nextPutAllBytes:` *aCharacterCollection*
(Subclass responsibility, WriteStream only.) Inserts the byte contents of *aCharacterCollection* as the next elements that the receiver can access. Returns *aCharacterCollection*.

`space` Adds a space to the output stream.

`tab` Adds a tab to the output stream.

Enumerating

`do:` *aBlock*	Evaluates the one-argument block *aBlock* for each of the remaining objects that the receiver can access.

Testing

`atEnd`	(Subclass responsibility.) Returns true if the receiver cannot access any more objects, false if it can.
`isExternal`	Returns true if the source of the receiver's information is external to the image, and false otherwise.

String

Instances of class String are indexed collections of Characters.

Superclasses CharacterCollection, SequenceableCollection, Collection, Object

Named Instance Variables None

Instance Format Byte, Indexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`at:` *anIndex* Returns the character at *anIndex*.

`numArgs` Returns the number of arguments the receiver would take, were the receiver a message selector.

`size` Returns the size of the receiver, in characters.

Adding

`addAll:` *aCharOrCharColl*

Equivalent to `add:` *aCharOrCharColl*.

`addAllBytes:` *aCharacterCollection*

Adds the byte contents of *aCharacterCollection* to the receiver. Returns *aCharacterCollection*.

The *aCharacterCollection* argument must be a kind of String or DoubleByteString.

`addLast:` *aCharOrCharColl*

Equivalent to `add:` *aCharOrCharColl*.

`insertAll:` *aCharOrCharColl* `at:` *anIndex*

Inserts *aCharOrCharColl* at the specified index. Returns *aCharOrCharColl*

Backward Compatibility

Methods in this category are obsolete and are provided only for compatibility with earlier releases of GemStone. They will be removed in a future release.

`toClientTextFile:` *fileName*

Obsolete in GemStone 4.1. Use an instance of GsFile to access the file system of the client or server machines.

Case-Insensitive Comparisons

< *aCharCollection*

Returns true if the receiver collates before the argument. Returns false otherwise.

The comparison is case-insensitive unless the receiver and argument are equal ignoring case, in which case upper case letters collate before lower case letters. The default behavior for SortedCollections and for the sortAscending method in UnorderedCollection is consistent with this method, and collates as follows:

```
#( c MM Mm mb mM mm x ) asSortedCollection
```

yields the following sort order:

```
c mb MM Mm mM mm x
```

<= *aCharCollection*

Returns true if the receiver collates before the argument or if all of the corresponding characters in the receiver and argument are equal. Returns false otherwise.

The comparison is consistent with that defined for the < method.

> *aCharCollection*

Returns true if the receiver collates after the argument. Returns false otherwise.

The comparison is consistent with that defined for the < method.

>= *aCharCollection*

Returns true if the receiver collates after the argument or if all of the corresponding characters in the receiver and argument are equal. Returns false otherwise.

The comparison is consistent with that defined for the < method.

`at:` *anIndex* `equalsNoCase:` *aCharCollection*

Returns true if *aCharCollection* is contained in the receiver, starting at *anIndex*. Returns false otherwise. The comparison is case-insensitive.

`equalsNoCase:` *aCharCollection*

Returns true if corresponding characters in the receiver and argument are equal and *aCharCollection* is comparable with the receiver. Returns false otherwise.

The comparison for equal is case-insensitive.

If aString is a Symbol, there is no difference in behavior (contrast with `String | =`).

`isEquivalent:` *aCharCollection*

Returns true if the receiver is equivalent to *aCharCollection*. The receiver and *aCharCollection* are compared without regard to case or internal representation of characters.

Case-Sensitive Comparisons

`=` *aCharCollection*

Returns true if corresponding characters in the receiver and argument are equal and *aCharCollection* is comparable with the receiver, and *aCharCollection* is not a kind of Symbol. Returns false otherwise.

The comparison for equal is case-sensitive.

Note that 'kind of Symbol' means either an instance of Symbol or instance of DoubleByteSymbol.

`at:` *anIndex* `equals:` *aCharCollection*

Returns true if *aCharCollection* is contained in the receiver, starting at *anIndex*. Returns false otherwise. The comparison is case-sensitive.

Case-Sensitive Searching

`includesValue:` *aCharacter*

Returns true if the receiver contains *aCharacter*, false otherwise. The search is case-sensitive.

`indexOf:` *aCharacter* `startingAt:` *startIndex*

Returns the index of the first occurrence of *aCharacter* in the receiver, not preceding *startIndex*. If the receiver does not contain *aCharacter*, returns zero.

The search is case-sensitive.

Concatenating

`,` *aCharOrCharCollection*

Returns a new instance of the receiver's class that contains the elements of the receiver followed by the elements of *aCharOrCharCollection*. The argument must be a String or an AbstractCharacter.

The result may not be an instance of the class of the receiver if one of the following rules applies:

1. If the receiver or argument is a kind of Symbol or ObsoleteInvariantString, the result is a String.
2. If the class of the argument is not String, and is ISOLatin or some other user-defined subclass of String, and is not a subclass of ObsoleteInvariantString, then the result is an instance of the class of the argument.
3. If the argument is a kind of DoubleByteString, the result is a DoubleByteString.

Warning: Creating a new instance and copying the receiver take time. If you can safely modify the receiver, it can be much faster to use the `addAll:` method. See the documentation of the Concatenating category of class SequenceableCollection for more details.

Converting

Separator characters in a String are used to define white space that separates words or substrings within the String from each other. These separators are used in conversion to break the String into its logical substrings. Separator characters are defined as those characters for which the Character>>isSeparator method returns true.

`->` *anObject*	Returns a SymbolAssociation with the receiver as the key and the given object as the value.
`asArrayOfKeywords`	Returns an Array of keyword substrings held by the receiver. The receiver is assumed to be a colon-separated list of substrings. These substrings are extracted and collected in an Array. If the receiver contains no colons, the Array will hold a copy of the receiver.
`asArrayOfPathTerms`	Returns an Array of path substrings held by the receiver. The receiver is assumed to be a period-separated list of substrings. These substrings are extracted and collected in an Array. If the receiver contains no periods, the Array will hold a copy of the receiver. Periods not meant to separate path terms may be escaped with a `$\` character.
`asArrayOfSubstrings`	Returns an Array of substrings held by the receiver. The receiver is assumed to be a separator-separated list of substrings. These substrings are extracted and collected in an Array. If the receiver contains no separators, the Array will hold a copy of the receiver. Separators not meant to separate substrings may be escaped with a `$\` character.
`asNumber`	Returns the receiver converted to a kind of number. If the receiver contains all digits (with optional radix notation), returns a kind of Integer. If the receiver has a slash, returns a Fraction. Otherwise conversion to a Float is attempted. An error may result if the receiver does not contain the proper format for a kind of Number.
`asSymbol`	Returns a Symbol that corresponds to the receiver.
`asSymbolKind`	Equivalent to `asSymbol`.

`asUppercase` Returns a new instance of the receiver's class, with all lowercase characters in the receiver changed to uppercase. If the receiver is a Symbol, returns an instance of String.

`copyWrappedTo:` *rightMargin*
Returns a String with the receiver's contents word-wrapped to the given right margin.

Copying

`copyFrom:` *index1* `to:` *index2* `into:` *anObject* `startingAt:` *index3*
Copies the elements of the receiver between *index1* and *index2*, inclusive, into *anObject* starting at *index3*, overwriting the previous contents. If *anObject* is the same object as the receiver, the source and destination blocks may overlap.

`copyReplaceAll:` *subString* `with:` *newSubString*
Returns a copy of the receiver with all occurrences of the given substring replaced with the *newSubString*.

`replaceFrom:` *start* `to:` *stop* `with:` *str* `startingAt:` *stridx*
Replaces the contents of the receiver from the given start index to the given stop index with the elements of the given string starting at the given index, *stridx*. A one-for-one replacement is performed, so the receiver will not change in size.

The search is case-sensitive.

`withCRs` Supplied for Smalltalk-80 compatibility. This is equivalent to `withLFs`.

`withLFs` Returns a copy of the receiver with all back-slashes replaced by line-feeds.

Execution

`evaluate` Compiles the receiver as an unbound method and executes it using the current default symbol list.

`evaluateInContext:` *anObject* `symbolList:` *aSymbolList*
Compiles the receiver as an instance method for the class of *anObject*, using *aSymbolList* as the symbol list. Executes the resulting GsMethod using *anObject* as self and returns the result of the execution. Generates an error if compilation errors occur.

Formatting

`asString` Returns the receiver.

`describeClassName` Returns a copy of the receiver with the Character $a prepended to the receiver's contents. This method is used for formatting class names in object descriptions, where the receiver is a string containing the name of a class. For example, the String UserClass, when sent the message `describeClassName`, returns aUserClass.

`linesIndentedBy:` *anInt*
Returns a copy of the receiver in which all lines have been indented by *anInt* spaces.

`printOn:` *aStream* Puts a displayable representation of the receiver on the given stream.

`printString` Returns a String whose contents are a displayable representation of the receiver.

`trimWhiteSpace` Returns a copy of the receiver with leading and trailing white space removed.

Other Comparisons

`asciiLessThan:` *aString* Returns true if the receiver collates before the argument using the ASCII collating table, which collates AB...Z...ab..z.

`equals:` *aString* `collatingTable:` *aByteArray*

Returns true if the receiver collates the same as the argument.

The collating sequence is defined by *aByteArray*, which must be a ByteArray of size 256.

aString must be a String or a DoubleByteString; if a DoubleByteString, then characters with value > 255 are collated with their native order.

If *aString* is a Symbol, there is no difference in behavior (constrast to String>>=) .

`greaterThan:` *aString* `collatingTable:` *aByteArray*

Returns true if the receiver collates after the argument.

The collating sequence is defined by *aByteArray*, which must be a ByteArray of size 256.

aString must be a String or a DoubleByteString; if a DoubleByteString, then characters with value > 255 are collated with their native order.

`greaterThanOrEqual:` *aString* `collatingTable:` *aByteArray*

Returns true if the receiver collates after or the same as the argument.

The collating sequence is defined by *aByteArray*, which must be a ByteArray of size 256.

aString must be a String or a DoubleByteString; if a DoubleByteString, then characters with value > 255 are collated with their native order.

`lessThan:` *aString* `collatingTable:` *aByteArray*

Returns true if the receiver collates before the argument.

The collating sequence is defined by *aByteArray*, which must be a ByteArray of size 256.

aString must be a String or a DoubleByteString; if a DoubleByteString, then characters with value > 255 are collated with their native order.

`lessThanOrEqual:` *aString* `collatingTable:` *anArray*

Returns true if the receiver collates before or the same as the argument as defined by the collating table *anArray*.

The collating sequence is defined by aByteArray, which must be a ByteArray of size 256.

aString must be a String or a DoubleByteString; if a DoubleByteString, then characters with value > 255 are collated with their native order.

Searching

`speciesForCollect` Returns a class, an instance of which should be used as the result of collect: or other projections applied to the receiver.

Storing and Loading

`loadFrom:` *passiveObj* Reads from *passiveObj* the passive form of an object. Converts the object to its active form by loading the information into the receiver.

`writeTo:` *passiveObj* Converts the receiver to its passive form and writes that information on *passiveObj*.

Testing

Separator characters in a String are used to define white space that separates words or substrings within the String from each other. These separators are used in conversion to break the String into its logical substrings. Separator characters are defined as those characters for which the `Character>>isSeparator` method returns true.

`containsSeparator` Returns true if the receiver contains a separator character.

`isDigits` Returns true if the receiver contains only digits. Returns false if the receiver contains non-digit characters.

`isInfix` Returns true if the receiver is an infix (binary) selector. Returns false otherwise.

`isKeyword` Returns true if the receiver is a keyword. Returns false otherwise.

`isMinusAndDigits` Returns true if the receiver contains a minus sign followed only by digits. Returns false if the receiver has any other characters.

`isValidIdentifier` Returns true if the receiver is a valid GemStone Smalltalk variable name, and false otherwise.

Updating

`at:` *anIndex* `put:` *aChar* Stores *aChar* at the specified location.

`lf` Appends a line-feed to the receiver.

`size:` *anInteger* Changes the size of the receiver to *anInteger*.

If *anInteger* is less than the current size of the receiver, the receiver is shrunk accordingly. If *anInteger* is greater than the current size of the receiver, the receiver is extended and new elements are initialized to nil.

`space` Appends a space to the receiver.

`tab` Appends a tab to the receiver.

Class Protocol

Backward Compatibility

Methods in this category are obsolete and are provided only for compatibility with earlier releases of GemStone. They will be removed in a future release.

`fromClientTextFile:` *fileName*

Obsolete in GemStone 4.1. Use an instance of GsFile to access the file system of the client or server machines.

Instance Creation

`withAll:` *aString* Returns an instance of the receiver containing the elements of the argument. If *aString* is a DoubleByteString, returns an instance of DoubleByteString.

Storing and Loading

`loadFrom:` *passiveObj* Reads from *passiveObj* the passive form of an object. Converts the object to its active form by loading the information into a new instance of the receiver. Returns the new instance.

StringKeyValueDictionary

A StringKeyValueDictionary is a KeyValueDictionary in which the keys are Strings or DoubleByteStrings. The hash function treats the strings as case-sensitive.

The hashing algorithm is described in

Pearson, Peter K., "Fast Hashing of Variable-Length Text Strings," Communications of the ACM, 33:6 (June 1990), 667-680.

The implementation employs two modifications:

- The hash function uses at most 2008 bytes of the string. Strings that are identical within this range have the same hash value.
- In his paper, Pearson describes a technique for producing a larger number of hash values by always computing the hash function to 24 bits. The hash value is the result of this function modulo the table size. Therefore, there is no value in specifying a table size greater than 2 to the 24th power.

Superclasses KeyValueDictionary, AbstractDictionary, Collection, Object

Named Instance Variables None

Instance Format Pointer, Indexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`at:` *aKey* `ifAbsent:` *aBlock*

Returns the value that corresponds to *aKey*. If no such key/value pair exists, returns the result of evaluating the zero-argument block *aBlock*.

`at:` *aKey* `otherwise:` *aValue*

Returns the value that corresponds to *aKey*. If no such key/value pair exists, returns the given alternate value.

`name` Returns the key of a key/value pair whose value is the receiver. If the receiver contains no such Association, returns nil.

Updating

`at:` *aKey* `put:` *aValue*

Stores the aKey/aValue pair in the receiver. Rebuilds the hash table if the addition caused the number of collisions to exceed the limit allowed. Returns *aValue*.

If *aKey* is being added for the first time, an invariant copy of it is stored as the key.

StringPair

A StringPair is an Association whose key and value are both constrained to be Strings.

Superclasses	Association, Object
Named Instance Variables	None
Instance Format	Pointer, Nonindexable, Variant
Subclass Creation	Allowed

StringPairSet

A StringPairSet is an IdentitySet that contains only StringPair objects.

Superclasses	IdentitySet, IdentityBag, UnorderedCollection, Collection, Object
Named Instance Variables	None
Instance Format	Nsc, Nonindexable, Variant
Subclass Creation	Allowed

Instance Protocol

Sorting

`sortAscending`	Returns an Array of the contents of the receiver sorted by key in ascending order.

Symbol

A Symbol is an invariant String for which all comparisons are case-sensitive. Symbols are used internally to represent variable names and selectors. Symbols are always invariant and cannot be modified at any time after they are created. Hence, the new and new: methods are disallowed.

All Symbols and DoubleByteSymbols are canonical, which means that it is not possible to create two of them that have the same value. If two canonical symbols compare as equal, then they are the same (identical) object. Every instance of DoubleByteSymbol will contain at least one Character whose value is greater than 255. A Symbol whose character values are all less than 256 is always an instance of Symbol.

GemStone places all canonical symbols in the DataCuratorSegment. However, GemStone does permit you to commit a canonical Symbol, even if you have no explicit write authorization for the DataCuratorSegment. GemStone also gathers all canonical symbols into one collection (a CanonicalStringDictionary) called AllSymbols, which it also places in the DataCuratorSegment.

Since canonical symbols are universally visible, it is not recommended that they be used for names that should remain private or secure. Such objects should be instances of InvariantString instead.

Since canonical symbols must be universally available, you cannot lock a Symbol or DoubleByteSymbol.

Since each canonical symbol has a unique value, you cannot copy a Symbol or DoubleByteSymbol. In addition, to guarantee canonicalization, you cannot send the become: or changeClassTo: messages to a Symbol or DoubleByteSymbol.

DoubleByteSymbol is in the classHistory of Symbol, so instances of DoubleByteSymbol may be stored into instance variables that are constrained to hold instances of Symbol. The inverse is not true, so you should always express symbol constraints as Symbol.

EUCSymbols are not canonicalized and cannot be used interchangeably with canonical symbols. They do not satisfy a constraint of Symbol, and are not accepted by the virtual machine as message selectors.

Superclasses	String, CharacterCollection, SequenceableCollection, Collection, Object
Named Instance Variables	None
Instance Format	Byte, Indexable, Variant

Subclass Creation Disallowed

Instance Protocol

Clustering

`cluster` Cluster the receiver in the current default cluster bucket, as long as the current user has write authorization for the segment of the receiver. Returns true.

Comparing

`=` *aCharCollection* Returns true if and only if the receiver and *aCharCollection* are identical, unless *aCharCollection* is an instance of ObsoleteSymbol. In that case, this method returns true if the receiver and *aCharCollection* are equivalent (in the sense of the `String | =` method). Returns false otherwise.

`hash` Returns a numeric hash key for the receiver.

`~=` *aCharCollection* This method can be optimized for Symbols since they are canonical.

Concatenating

`,` *aCharOrCharCollection* Returns a new instance of String that contains the elements of the receiver followed by the elements of *aCharOrCharCollection*. A String is returned rather than a Symbol to avoid the expense of unnecessary creation and canonicalization of Symbols.

Converting

`argumentCount` Interpreting the receiver as a message selector, returns the number of arguments that the selector requires.

`asSymbol` Returns the receiver.

`asSymbolKind` Equivalent to `asSymbol`.

`copy` Returns self, copies of canonical symbols are not allowed.

`keywords` Returns a collection of the keywords in the receiver, assuming that the receiver is a keyword message selector.

Formatting

`asString`	Returns a copy of the receiver as an instance of class String.
`printOn:` *aStream*	Puts a displayable representation of the receiver on the given stream. That representation conforms to GemStone Smalltalk parsing rules.
`withNoColons`	Returns a String containing the value of the receiver with all colons removed. A String is returned rather than a Symbol to avoid the expense of unnecessary creation and canonicalization of Symbols.

Testing

`isSymbol`	Returns true.
`precedence`	Returns the precedence of the receiver, were it a message selector, with 1=unary, 2=binary and 3=keyword.

Class Protocol

Instance Creation

`new`	Disallowed. To create a new Symbol, use the class method `withAll:` instead.
`new:` *size*	Disallowed. To create a new Symbol, use the class method `withAll:` instead.
`withAll:` *aString*	Returns a canonical symbol that has the same Characters as *aString*. Returns an existing canonical symbol if it is already in AllSymbols, or if it was created earlier in the current session. Otherwise, creates and returns a new canonical symbol. The canonical symbol that this method returns is always a Symbol, never a DoubleByteSymbol.

Storing and Loading

`loadFrom:` *passiveObj*	Reads from *passiveObj* the passive form of an object. Converts the object to its active form by loading the information into a new instance of the receiver. Returns the new instance.

SymbolAssociation

A SymbolAssociation is an Association whose key is constrained to be a canonical symbol (Symbol or DoubleByteSymbol).

Superclasses	Association, Object
Named Instance Variables	None
Instance Format	Pointer, Nonindexable, Variant
Subclass Creation	Allowed

SymbolDictionary

A SymbolDictionary is an IdentityDictionary in which the keys are canonical symbols (Symbols or DoubleByteSymbols) and the values are SymbolAssociations. The key of each SymbolAssociation is also the key used by the SymbolDictionary to access that SymbolAssociation.

Only SymbolDictionaries can be used in symbol lists.

Superclasses IdentityDictionary, IdentityKeyValueDictionary, KeyValueDictionary, AbstractDictionary, Collection, Object

Named Instance Variables None

Instance Format Pointer, Indexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`associationAt:` *aKey* Returns the SymbolAssociation with key *aKey*. Generates an error if no such SymbolAssociation exists.

`associationAt:` *aKey* `ifAbsent:` *aBlock*
Returns the SymbolAssociation with key *aKey*. If no such SymbolAssociation exists, returns the result of evaluating the zero-argument block *aBlock*.

`associationAt:` *aKey* `otherwise:` *defaultValue*
Returns the SymbolAssociation with key *aKey*. If no such SymbolAssociation exists, returns the given default value.

`at:` *aKey* Returns the value at the given key. Generates an error if *aKey* not found.

`at:` *aKey* `ifAbsent:` *aBlock*
Returns the value of the SymbolAssociation with key *aKey*. If no such SymbolAssociation exists, returns the result of evaluating the zero-argument block *aBlock*.

`at:` *aKey* `otherwise:` *defaultValue*
: Returns the value at the given key. If *aKey* is not found, returns *defaultValue*.

`keys`
: Returns a SymbolSet containing the receiver's keys.

`name`
: Returns the key of a SymbolAssociation whose value is the receiver. If the receiver contains no such SymbolAssociation, returns nil.

`name:` *aSymbol*
: Equivalent to `self at:` *aSymbol* `put: self`.

`names`
: Returns an Array that contains all the keys of entries in the receiver whose value is the receiver itself. The order of the elements in the result is arbitrary. If no such keys are found, returns an empty Array.

Backward Compatibility

Methods in this category are obsolete and are provided only for compatibility with earlier releases of GemStone. They will be removed in a future release.

`detectValues:` *aBlock* `ifNone:` *exceptionBlock*
: Obsolete in GemStone 5.0. Use the keysAndValuesDo: method instead.

Evaluation

`textForError:` *aNumber* `args:` *anArray*
: Returns a String representing the given error.

Formatting

`printOn:` *aStream*
: Puts a displayable representation of the receiver on the given stream.

Searching

`includesKey:` *aKey*
: Reimplemented from KeyValueDictionary for efficiency.

Updating

`addAssociation:` *aSymbolAssociation*
Add the argument to the receiver.

`at:` *aKey* `put:` *aValue*
If the receiver already contains a SymbolAssociation with the given key, this makes *aValue* the value of that SymbolAssociation. Otherwise, this creates a new SymbolAssociation with the given key and value and adds it to the receiver. *aKey* must be a Symbol. Returns *aValue*.

`atHash:` *hashIndex* `putKey:` *aKey*
Updates the hash table by storing *aKey* at the specified *hashIndex*.

`atHash:` *hashIndex* `putValue:` *aValue*
Updates the hash table by storing *aValue* at the specified *hashIndex*.

`renameAssociationFrom:` *key1* `to:` *key2*
Lookup the association in the receiver with having *key1*, and change its key to *key2*. Raises an error if *key1* is not found, or if *key2* already exists. *key1* and *key2* must be Symbols.

`swapKey:` *key1* `with:` *key2*
In the receiver, lookup the Associations for *key1* and *key2* and swap the keys of the two Associations. Returns the receiver. If either *key1* or *key2* is not found in the receiver, raises an error.

SymbolKeyValueDictionary

A SymbolKeyValueDictionary is an IdentityKeyValueDictionary in which the keys are canonical symbols (Symbols or DoubleByteSymbols). The separate implementation is necessary in order to support canonicalization of the symbols that are used as keys.

SymbolKeyValueDictionaries cannot be used in symbol lists.

Superclasses IdentityKeyValueDictionary, KeyValueDictionary, AbstractDictionary, Collection, Object

Named Instance Variables None

Instance Format Pointer, Indexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`keys` Returns a SymbolSet containing the receiver's keys.

Updating

`at:` *aKey* `put:` *aValue* Stores the aKey/aValue pair in the hash dictionary. *aKey* must be convertable to a Symbol.

Rebuilds the hash table if the addition caused the number of collisions to exceed the limit allowed.

SymbolList

A SymbolList is an Array whose elements are instances of SymbolDictionary.

It is used in compilation of GemStone Smalltalk code, in order to resolve references to objects by name. Given a Symbol as a name, the SymbolList searches its dictionary elements in order, and the first key that matches the given Symbol then resolves to the object that is the value at that key in that dictionary.

Superclasses Array, SequenceableCollection, Collection, Object

Named Instance Variables None

Instance Format Pointer, Indexable, Variant

Subclass Creation Disallowed

Instance Protocol

Accessing

`names` Returns an Array of Strings containing the names of the receiver's SymbolDictionaries.

This method assumes that each SymbolDictionary in the receiver contains a SymbolAssociation whose value is that SymbolDictionary. If any SymbolDictionary does not contain such a SymbolAssociation, it is represented in the result Array as '(unnamed Dictionary)'.

Formatting

`namesReport` Returns a formatted String describing the position and name of each Dictionary in the receiver's symbol list.

This method assumes that each Dictionary in the symbol list contains an Association whose value is that Dictionary. If any Dictionary does not contain such an Association, it is represented in the result String as '(unnamed Dictionary)'.

Searching

`dictionaryAndSymbolOf:` *anObject*

Returns the Dictionary that names *anObject*, and also returns the name which that Dictionary associates with *anObject*. More precisely, this returns an Array containing two elements:

- The Dictionary in the receiver that contains an Association whose value is *anObject*.
- The Symbol which is that Association's key.

The receiver is searched in the same order that the compiler searches it. (For more information about symbol resolution, see the *GemStone Programming Guide*.) If *anObject* is not found in the receiver, returns nil.

`objectNamed:` *aSymbol* Returns the first object in the receiver that has the given name. If no object is found with the given name, returns nil.

`resolveSymbol:` *aString* Searches the receiver for an Association whose key is equal to *aString*, and returns that Association. If no such Association is found, returns nil.

`symbolResolutionOf:` *aString*

Searches the receiver for *aString*. If *aString* is found, returns a formatted String that describs the position in the receiver of the Dictionary defining *aString*, the name of that Dictionary, and *aString*.

Generates an error if *aString* is not defined in the receiver.

Updating

`createDictionaryNamed:` *dictName* `at:` *anIndex*

Creates a new SymbolDictionary in the Segment of the receiver. Adds to the dictionary an Association whose key is *dictName* and whose value is the dictionary. Inserts the dictionary in the receiver at *anIndex*.

If *anIndex* is less than 1, the dictionary is inserted at the front of the list. If *anIndex* is greater than the size of the list, it is inserted at the end of the list. If the receiver already contains a dictionary named *dictName*, raises an error.

`removeDictionaryNamed:` *aSymbol*

Removes the first dictionary found in the receiver that contains an Association whose key is *aSymbol* and whose value is the dictionary. Returns the removed dictionary.

If no such dictionary is found, raises a KeyNotFound error.

`removeDictionaryNamed:` *aSymbol* `ifAbsent:` *aBlock*

Removes the first dictionary found in the receiver that contains an Association whose key is *aSymbol* and whose value is the dictionary. Returns the removed dictionary.

If no such dictionary is found, returns the result of evaluating the zero-argument Block *aBlock*.

`replaceElementsFrom:` *aSymbolList*

Removes all elements in the receiver and inserts all elements of *aSymbolList* in it, in the same order as in *aSymbolList*.

If the argument is not a SymbolList, raises an error.

SymbolSet

A SymbolSet is an IdentitySet whose elements must be canonical symbols (Symbols or DoubleByteSymbols).

Superclasses	IdentitySet, IdentityBag, UnorderedCollection, Collection, Object
Named Instance Variables	None
Instance Format	Nsc, Nonindexable, Variant
Subclass Creation	Allowed

System

System is an abstract class that has no instances. It implements class methods for object locking and for operations that are usually found in traditional operating systems. The data curator may restrict user access to these messages. For an explanation of the role of the data curator, refer to your *GemStone System Administration Guide*.

Superclasses Object

Named Instance Variables None

Instance Format Pointer, Nonindexable, Variant

Subclass Creation Disallowed

Class Protocol

Authorization

`canRead:` *anObject*
This method tests whether the user has authorization to read *anObject* without adding it to the readSet and returns a Boolean result.

`canWrite:` *anObject*
This method tests whether the user has authorization to write *anObject* without adding it to the readSet or writeSet and returns a Boolean result.

Backward Compatibility

Methods in this category are obsolete and are provided only for compatibility with earlier releases of GemStone. They will be removed in a future release.

`clusterBucket`
Obsolete in GemStone 3.2.

`contentsOfServerDirectory:` *aSpecString*
Obsolete in GemStone 5.0. Use the `GsFile>>contentsofDirectory:onClient:` method instead.

`deleteServerFile:` *aFileSpec*
Obsolete in GemStone 5.0. Use the `GsFile>>removeServerFile:` method instead.

`exclusiveLock:` *anObject* `ifDenied:` *denyBlock* `ifChanged:` *changeBlock* `ifNotCommitted:` *notCommittedBlock*

Obsolete in GemStone 4.0. Locks are now allowed on objects that have not been committed.

`exclusiveLockObjAndIndexes:` *anObject*

Obsolete in GemStone 4.0. Indexes now have reduced conflict behavior and locks are now allowed on objects that have not been committed.

`exclusiveLockObjAndIndexes:` *anObject* `ifDenied:` *denyBlock* `ifChanged:` *changeBlock* `ifNotCommitted:` *notCommittedBlock*

Obsolete in GemStone 4.0. Indexes now have reduced conflict behavior and locks are now allowed on objects that have not been committed.

`gemStatistics` Obsolete in GemStone 4.1. Use the `pageReads` and `pageWrites` methods instead for I/O statistics.

`myUserGlobals` Obsolete in GemStone 5.0.

`readLock:` *anObject* `ifDenied:` *denyBlock* `ifChanged:` *changeBlock* `ifNotCommitted:` *notCommittedBlock*

Obsolete in GemStone 4.0. Locks are now allowed on objects that have not been committed.

`readLockObjAndIndexes:` *anObject*

Obsolete in GemStone 4.0. Indexes now have reduced conflict behavior and locks are now allowed on objects that have not been committed.

`readLockObjAndIndexes:` *anObject* `ifDenied:` *denyBlock* `ifChanged:` *changeBlock* `ifNotCommitted:` *notCommittedBlock*

Obsolete in GemStone 4.0. Indexes now have reduced conflict behavior and locks are now allowed on objects that have not been committed.

`removeLockAllNoErr:` *aCollection*

Obsolete in GemStone 4.0. Use the `removeLockAll:` method instead.

`removeLockNoErr:` *anObject*

Obsolete in GemStone 4.0. Use the `removeLockAll:` method instead.

`signalFromGemStoneSession`
Obsolete in GemStone 5.0.

`stoneStatistics` Obsolete in GemStone 4.1. Use the `pageReads` and `pageWrites` methods instead for I/O statistics.

`writeLock:` *anObject* `ifDenied:` *denyBlock* `ifChanged:` *changeBlock* `ifNotCommitted:` *notCommittedBlock*
Obsolete in GemStone 4.0. Locks are now allowed on objects that have not been committed.

`writeLockObjAndIndexes:` *anObject*
Obsolete in GemStone 4.0. Indexes now have reduced conflict behavior and locks are now allowed on objects that have not been committed.

`writeLockObjAndIndexes:` *anObject* `ifDenied:` *denyBlock* `ifChanged:` *changeBlock* `ifNotCommitted:` *notCommittedBlock*
Obsolete in GemStone 4.0. Indexes now have reduced conflict behavior and locks are now allowed on objects that have not been committed.

Clustering

These methods are used in managing cluster buckets, the streams of disk pages in which objects are congregated during clustering. Clustering is explained in the *GemStone Programming Guide*.

`clusterAllSymbols` This method clusters the AllSymbols hash dictionary and all of the symbols to which it refers.

`clusterBucket:` *aClusterBucketOrId*
This method sets the current default ClusterBucket to the ClusterBucket with the specified clusterId. The argument may be an instance of ClusterBucket, or a positive SmallInteger which specifies an instance of ClusterBucket.

`currentClusterBucket` Returns the instance of ClusterBucket that is the current default.

`currentClusterId` This method returns a SmallInteger which is the id of the ClusterBucket that is the current default bucket.

`maxClusterBucket` Returns the maximum legal clusterId as a SmallInteger.

Configuration File Access

`configurationAt:` *aName*

Returns the value of the specified configuration file parameter, giving preference to the Gem process if the parameter applies to the Gem.

`gemConfigurationAt:` *aName*

Returns the value of the specified configuration file parameter from the current session. Returns nil if that parameter is not applicable to a Gem.

`gemConfigurationReport`

Returns a SymbolDictionary whose keys are the names of configuration file parameters, and whose values are the current settings of those parameters in the current session's Gem process. Parameters that are not applicable to gem and those that are undefined are not included in the result.

`stoneConfigurationAt:` *aName*

Returns the value of the specified configuration file parameter from the repository monitor process (stone). Returns nil if that parameter is not applicable to the stone.

`stoneConfigurationAt:` *aName* `put:` *aValue*

Changes the value of the specified stone configuration parameter.

See comments in the method `configurationAt:put:` for complete documentation.

`stoneConfigurationReport`

Returns a SymbolDictionary whose keys are the names of configuration file parameters, and whose values are the current settings of those parameters in the repository monitor process (stone). Parameters that are not applicable to stone and those that are undefined are not include in the result.

Debugging Support

`stackDepth` Returns current depth of the GemStone Smalltalk stack.

`stackDepthHighwater` Returns largest depth of the GemStone Smalltalk stack since session login.

`stackLimit` Returns the approximate limit on the depth of the GemStone Smalltalk stack. The stack size is determined by the configuration file parameter GEM_MAX_SMALLTALK_STACK_DEPTH.

`stackLimit:` *anInteger* Has no effect in GemStone 5.0. Provided for compatibility.

Disk Space Management

`findObjectsLargerThan:` *aSize* `limit:` *aLimit*

Searches GemStone for objects larger than *aSize*, and returns an Array of any such objects. The search continues until all such objects have been found, or until the size of the result reaches the specified maximum *aLimit*. Both *aSize* and *aLimit* must be positive SmallIntegers.

The result contains only those objects which reside within segments that the user is authorized to read. If this method encounters an object larger than *aSize* which it is not authorized to read, the final element of the result will be the String 'Read Authorization Error Encountered'.

The result contains both permanent and temporary objects. The temporary objects found may vary from run to run.

Note that this method may take a considerable length of time to execute.

Environment Access

`clientEnvironmentVariable:` *varName*

Expands the environment variable named *varName* in the GemBuilder for C client process, returning a String. The *varName* argument should be a kind of String.

Returns nil if any of the following are true:

- *varName* is not a byte format object.
- There is no environment variable defined with name *varName*.
- The value of the environment variable is more than approximately 8000 bytes.
- The size of *varName* exceeds approximately 8000 bytes.

`gemEnvironmentVariable:` *varName*

Expands the environment variable named *varName* in the Gem process, returning a String. *varName* should be a kind of String.

Returns nil if any of the following are true:

- *varName* is not a byte format object.
- There is no environment variable defined with name *varName*.
- The value of the environment variable is more than approximately 8000 bytes.
- The size of *varName* exceeds approximately 8000 bytes.

`sessionPerformingBackup`

Returns the session id of the session that is performing a backup. If there is no such session, returns -1.

`stoneName` Returns a Symbol whose value is the full network name of the stone to which this session is logged in.

Error Handling

`genericSignal:` *errIdentifier* `text:` *aString*

Raise a user-defined signal with no arguments.

The argument *errIdentifier* is a user-defined object, to distinguish user errors, and may be nil. The argument *aString* appears in GemStone's error message for this error, and may be nil.

`genericSignal:` *errIdentifier* `text:` *aString* `arg:` *anArg*

Raise a user-defined signal with one argument.

The argument *errIdentifier* is a user-defined object, to distinguish user errors, and may be nil. The argument *aString* appears in GemStone's error message for this error, and may be nil. The argument *anArg* appears as the third argument to the error.

`genericSignal:` *errIdentifier* `text:` *aString* `args:` *anArray*

Raise a user-defined signal.

The argument *errIdentifier* is a user-defined object, to distinguish user errors, and may be nil. The argument *aString* appears in GemStone's error message for this error, and may be nil. The argument *anArray* appears as the third argument to the error.

`signal:` *anInteger* `args:` *anArray* `signalDictionary:` *anErrorDict*

This method generates the specified signal (or error), along with its associated arguments. If an Exception is available to field the error, the Exception is invoked, otherwise returns control to the controlling GemBuilder for C (GCI) interface.

Hidden Set Support

`HiddenSetSpecifiers` Returns a list of the hiddenSet specifiers.

Host System Access

In each of these file system access methods, it is best to specify the full pathname of the server text file in the method's argument.

Under Unix, be sure that the case of the argument matches the case of the Unix file name; Unix is case-sensitive.

Also note that under Unix, each of these methods inherits environment variables from the GemStone session process, rather than from your user session. In addition, the method `performOnServer:` invokes the Bourne shell, even if you use a different login shell. For these reasons, you might want to avoid using environment variables in the arguments to these methods.

`performOnServer:` *aString*
: This method causes the operating system commands in *aString* to be executed as a spawned subprocess. Generates an error if *aString* cannot be executed by the operating system.

  Under Unix, commands in *aString* can have exactly the same form as a shell script. For example, newlines or semicolons can separate commands, and a backslash can be used as an escape character.

Instance Creation

`new`
: Disallowed. You may not create new instances of System.

Lock Status

`lockKind:` *anObject*
: Returns a Symbol (#none, #read, #write, or #exclusive) representing the kind of lock held on *anObject* by any session in the system.

`lockOwners:` *anObject*
: Returns an Array of session numbers (SmallIntegers) representing the sessions that hold a lock on *anObject*. If the object is not locked by any session, the result Array is empty. Note that a write or exclusive lock can have only one owner.

`lockStatus:` *anObject*

Returns a two-element Array, where the first element is a Symbol representing the kind of lock held on *anObject* (#none, #read, #write, or #exclusive) and the second element is an Array of session numbers (SmallIntegers) representing the sessions that hold the lock.

If there are no locks on *anObject*, the first element is the Symbol #none and the second element is an empty Array.

Only locks on permanent objects are reported.

`myLockKind:` *anObject*

Returns a Symbol that indicates what kind of lock the current session has on *anObject*: one of #none, #read, #write, or #exclusive.

`sessionLocks`

Returns a three-element Array describing the locks held by the current session. The first element is an Array of all read-locked objects, the second is an Array of all write-locked objects, and the third is an Array of all exclusive-locked objects. If the current session holds no locks of a particular kind (read, write, exclusive), then the corresponding Array is empty. If the current session holds no locks at all, then all three of these Arrays are empty.

`systemLocks`

Returns a Dictionary describing all objects that are currently locked. For each Association in the result Dictionary, the key is a SmallInteger (the session number of a GemStone session that holds locks) and the value is the three-element Array described in the `sessionLocks` method. If no sessions hold any locks, the result Dictionary is empty.

The Arrays in the result Dictionary contain only those objects that are visible to the current session. This method does not return locks that the current session cannot see (objects that have been committed since the beginning of the current transaction, uncommitted objects from other sessions, or locks on objects for which this session has no read authorization).

Locks on temporary objects in other sessions are not reported.

Notification

`addAllToNotifySet:` *aCollection*

Add all the elements of *aCollection* to the notify set. Special objects and uncommitted objects are not permitted, since neither of these can be modified by other sessions.

`addToNotifySet:` *anObject*

Add *anObject* to the notify set. The argument *anObject* cannot be a special object nor an uncommitted object, since neither can be modified by other sessions.

`clearNotifySet` Remove all of the objects that are currently in the notify set.

`disableSignaledGemStoneSessionError`

Set the current GemStone session so that it cannot receive signals from other GemStone sessions.

`disableSignaledObjectsError`

Disable the generation of an error when a member of the notify set is added to the signaled objects set.

`enableSignaledGemStoneSessionError`

Enable the current GemStone session to receive signals from other GemStone sessions. One GemStone session receives a signal from another session when a RT_ERR_SIGNAL_GEMSTONE_SESSION exception is raised.

The receiving session processes the signal with an exception handler. When GemStone raises one signal exception, it also disables further signal exceptions, to allow the exception handler to run without receiving another interrupt. The exception handler should therefore re-enable signal exceptions when it is done with its other processing.

A signal is not exactly an interrupt, and it does not automatically awaken an idle session. Both the GemStone Smalltalk virtual machine and GemBuilder for C can raise the signal exception. But the process of the session must activate the virtual machine or interface before the signal can be received.

`enableSignaledObjectsError`

Enable the generation of an error when a member of the notify set is added to the signaled objects set. When this error (RT_ERR_SIGNAL_COMMIT) is signaled, it is also disabled to allow the exception handler to run without receiving another interrupt. Therefore, the exception handler should re-enable the condition.

`notifySet` Returns an array of the objects that the user has registered for notification when a new state is committed.

`removeAllFromNotifySet:` *aCollection*

Removes all elements of *aCollection* from the notify set. Does not generate an error if any of the elements are not in the notify set.

`removeFromNotifySet:` *anObject*

Removes *anObject* from the notify set. Does not generate an error if *anObject* is not in the notify set.

`signaledGemStoneSessionErrorStatus`

Returns true to indicate that the current GemStone session can receive signals from other GemStone sessions. Returns false otherwise.

`signaledObjects` Returns an Array containing the objects that have been signaled since the last time this method was executed. The elements in the Array are a subset of the notify set. Clear the set of signaled objects.

`signaledObjectsErrorStatus`

Returns true to indicate that the system generates errors when objects are added to the signaled objects set. Returns false otherwise.

Performance Monitoring

`cacheStatistics:` *aProcessSlot*

Returns an Array whose contents are described by the result of the `cacheStatisticsDescription` method. The array contains statistics for the specified slot in the GemStone shared memory cache to which this session is attached.

The argument *aProcessSlot* should be a SmallInteger between 0 and the number of process slots in the shared cache minus 1, inclusive. If *aProcessSlot* is outside the range of valid process slots, or the session executing this method is not using a shared cache, generate an error. If the slot specified by *aProcessSlot* is an inactive slot, returns nil.

The process slots that are predefined are:

slot 0: The shared page cache monitor.

slot 1: The stone if the cache is on the same machine as the stone. Otherwise, a page server that is used to monitor the cache for the stone.

No other slots are guaranteed. However, slot 2 is the often the page server and slot 3 is often the Gcgem. These depend to some extent on the relative speed of the process(es) during startup. In addition, the Gcgem can be shut down, and when it is restarted, it is unlikely to end up at the same position.

`cacheStatisticsDescription`

Returns an Array of Strings describing the result of the method cacheStatistics.

`millisecondsToRun:` *aBlock*

Returns the number of CPU milliseconds used while evaluating *aBlock*. The argument *aBlock* must be a zero-argument block.

`myCacheProcessSlot` Returns the process slot in the SharedPageCache that corresponds to my process. If the SharedPageCache is not in use, returns -1.

`pageReads` Returns the number of Repository page read operations performed since the start of the Gem process. If the Gem is remote, this corresponds to reads performed in the current session. If the Gem is linked, it corresponds to reads performed since the application was invoked.

`pageWrites` Returns the number of Repository page write operations performed since the start of the Gem process. If the Gem is remote, this corresponds to writes performed in the current session. If the Gem is linked, it corresponds to writes performed since the application was invoked.

Reduced Conflict Support

`clearRcValueCache` Clears the cache of calculated values for reduced conflict classes by setting the temporary session state slot to nil.

`clearRedoLog` Clear the redo log by setting the temporary session state slot to nil.

Warning:
Clearing the redo log will probably prevent Reduced Conflict classes from resolving conflicts. Sending this message negates this capability for the current transaction.

This is a protected method.

`rcValueCache` Returns the cache dictionary that is stored in temporary session state used to hold calculated values for reduced conflict classes. If it does not exist, create it.

`rcValueCacheAt:` *aKey* `for:` *anObject* `ifAbsent:` *aBlock*
Returns the associated value at the given key for *anObject*. If the key is not present, execute the zero-argument block.

`rcValueCacheAt:` *aKey* `for:` *anObject* `otherwise:` *aValue*
Returns the associated value at the given key for *anObject*. If the key is not present, returns *aValue*.

`rcValueCacheAt:` *aKey* `otherwise:` *aValue*
Returns the associated value at the given key for anObject. If the key is not present, returns *aValue*.

`rcValueCacheAt:` *aKey* `put:` *aValue* `for:` *anObject*
Adds the given key/value pair for *anObject*. Returns the receiver.

`redoLog` Returns the redo log that is stored in the temporary session state. Create it if it does not exist.

Releasing Locks

GemStone maintains two sets of objects that you can manipulate with methods in this category. The commit release locks set contains locked objects whose locks will be released as part of the next successful commit operation. The commit-or-abort release locks set contains locked objects whose locks will be released as part of the next successful commit operation or abort operation. To gain complete control over the automatic releasing of locks at the end of a transaction, use these methods during the transaction to govern the membership of objects in these sets.

`addAllToCommitOrAbortReleaseLocksSet:` *aCollection*
Add each element of *aCollection* to the commit-or-abort release locks set. If an element of *aCollection* is not locked by the current session, then it is not added to the set.

`addAllToCommitReleaseLocksSet:` *aCollection*
Add each element of *aCollection* to the commit release locks set. If an element of *aCollection* is not locked by the current session, then that element is not added to the set.

`addToCommitOrAbortReleaseLocksSet:` *anObject*
Add *anObject* to to the commit-or-abort release locks set. If *anObject* is not locked by the current session, then it is not added to the set.

`addToCommitReleaseLocksSet:` *anObject*
Add *anObject* to the commit release locks set. If *anObject* is not locked by the current session, then it is not added to the set.

`clearCommitOrAbortReleaseLocksSet`
Remove all objects from the commit-or-abort release locks set.

`clearCommitReleaseLocksSet`
Remove all objects from the commit release locks set.

`commitOrAbortReleaseLocksSetIncludes:` *anObject*
Returns true if *anObject* is in the commit-or-abort release locks set. Returns false otherwise.

`commitReleaseLocksSetIncludes:` *anObject*

Returns true if *anObject* is in the commit release locks set. Returns false otherwise.

`removeAllFromCommitOrAbortReleaseLocksSet:` *aCollection*

Remove all the elements of *aCollection* from the commit-or-abort release locks set. If an element of *aCollection* is not a member of the set, it is ignored.

`removeAllFromCommitReleaseLocksSet:` *aCollection*

Remove all elements of *aCollection* from the commit release locks set. If an element of *aCollection* is not a member of the set, it is ignored.

`removeFromCommitOrAbortReleaseLocksSet:` *anObject*

Remove *anObject* from the commit-or-abort release locks set. If *anObject* is not a member of the set, do nothing.

`removeFromCommitReleaseLocksSet:` *anObject*

Remove *anObject* from the commit release locks set. If *anObject* is not a member of the set, do nothing.

Removing Locks

`removeLock:` *anObject* Removes the lock held by the current session on *anObject*. This method suceeds even if you do not have read authorization for *anObject*. Returns the receiver.

`removeLockAll:` *aCollection*

Removes all locks held by the current session on the objects in *aCollection*. If an object in *aCollection* is not locked by the current session, that object is ignored. Returns the receiver.

`removeLocksForSession`

Removes all locks held by this session. Returns the receiver. This method succeeds even if the session no longer has read authorization for one or more of its locked objects.

Runtime Configuration Access

`configurationAt:` *aName* `put:` *aValue*

Change the value of the specified configuration parameter.

The changeable parameters all require *aValue* to be a SmallInteger.

Configuration parameters should not be changed unless there is a clear reason for doing so, since incorrect settings of parameters can have serious adverse effects on GemStone performance.

Configuration parameters for stone that are transferred to Gem processes are only read by the Gem at login, so changes using this method to stone parameters may have no effect on existing sessions.

Parameters in the Gem with the following names may be changed by any user at any time:

- #NotConnectedThreshold
- #GemIOLimit
- #GemTempObjCacheSize
- #GemNativeCodeThreshold
- #GemNativeCodeMax
- #GemFreeFrameLimit
- #NotConnectedDelta

Parameters in the Gem with the following names may be changed only by users who have the correct privilege and who follow any other restrictions:

- #ConcurrencyMode - Requires SessionAccess privilege. The current session must be the only session logged in other than GcGem.
- #LoginsSuspended - Requires SystemControl privilege.
- #StnLogLoginFailureLimit - Requires OtherPassword privilege.
- #StnLogLoginFailureTimeLimit - Requires OtherPassword privilege.
- #StnDisableLoginFailureLimit - Requires OtherPassword privilege.

- #StnDisableLoginFailureTimeLimit - Requires OtherPassword privilege.

All other parameters in the Gem that are changeable at run time may be changed only by SystemUser, and should not be changed in the course of normal GemStone operation.

`gemConfigurationAt:` *aName* `put:` *aValue*

Changes the value of the specified gem configuration parameter.

See comments in the method `configurationAt:put:` for complete documentation.

Session Control

`clientIsRemote`

Returns true if the GemBuilder for C client for this session is in a different process than the Gem, otherwise returns false.

`currentSegment`

Returns the Segment in which objects created in the current session are stored. At login, the current segment is the default segment of the UserProfile for the session of the sender.

`currentSegment:` *aSegment*

Redefines the Segment in which subsequent objects created in the current session will be stored. Returns the receiver.

Exercise caution when executing this method. If, at the time you attempt to commit your transaction, you no longer have write authorization for *aSegment*, an error will be generated, and you will be placed back into your default Segment.

`currentSessionNames`

Returns a formatted String containing, for each current GemStone session, the session number and userId.

If any sessions (including the Gcgem) other than the current session are logged in, this method requires SessionAccess privilege.

`currentSessions`

Returns an array of SmallIntegers corresponding to all of the sessions currently running on the GemStone system.

`descriptionOfSession:` *aSessionId*

Returns a ten-element Array describing the session identified by *aSessionId*:

1. The UserProfile of the session.
2. The processId of the gem process of the session (an Integer).
3. The hostname of the machine running the gem process (a String, limited to 127 bytes).
4. Primitive number in which the gem is executing, or 0 if it is not executing in a long primitive.
5. Time of the session's most recent `beginTransaction`, `commitTransaction`, or `abortTransaction` (from `System timeGmt`).
6. The session state (a SmallInteger).
7. A Boolean whost value is true if the session is currently in a transaction, and false if it is not.
8. A Boolean whost value is true if the session is currently referencing the oldest commit record, and false if it is not.
9. The session's serial number (a SmallInteger).
10. The session's sessionId (a SmallInteger).

Because a session can update its commit record without committing a transaction, it is possible that no session actually references the oldest commit record. Therefore, the eighth element may be false for all current sessions.

To execute this method for any session other than your current session, you must have the SessionAccess privilege.

`descriptionOfSessionSerialNum:` *aSerialNumber*

Returns a ten-element Array describing the session identified by *aSerialNumber*.

See `System (C) | descriptionOfSession:` for documentation on the contents of the result Array.

Requires SessionAccess privilege if *aSerialNumber* is not the current session.

`maxSessionId` Returns a SmallInteger representing the maximum number of sessions allowed on the system based upon the stone configuration parameter.

`myUserProfile` Returns the UserProfile of the current session.

`session` Returns a SmallInteger representing the session of the sender.

`sessionsReferencingOldestCr`

Returns an Array containing the sessionIds of the sessions that are currently referencing the oldest commit record. Because a session can update its commit record without committing a transaction, it is possible that no session actually references the oldest commit record. Therefore, this method may return an empty Array.

`sleep:` *aTime* Sleep for *aTime* seconds. *aTime* must be a positive SmallInteger. If *aTime* is zero, this method has no effect.

This method is not currently interruptible.

`stopOtherSessions` Prevents any new sessions from being initiated; then, for each active session other than the session of the user executing this method, aborts the transaction and terminates that session. The also stops the Garbage Collector session.

To reenable logins, send the message `System resumeLogins`. Otherwise, logins are automatically reenabled when this session logs out.

To execute this method, you must have explicit privilege for SessionAccess and SystemControl in your UserProfile.

`stopSession:` *aSessionId*

Aborts the transaction of the specified session (a SmallInteger), then terminates that session. Returns the receiver. If the indicated session is not active, no operation is performed.

To execute this method, you must have explicit privilege from your system data curator.

`userProfileForSession:` *aSessionId*

Returns the UserProfile attached to the specified session (a SmallInteger). If the indicated session is not active, returns nil.

Requires SessionAccess privilege if *aSessionId* is not the current session.

`users`

Returns a Set of UserProfiles for all users known to the system.

Setting Locks

`exclusiveLock:` *anObject*

Requests an exclusive lock on *anObject*. This method denies an exclusive lock on an object under the following circumstances:

- Another session already holds any kind of lock to the object.
- You do not have write authorization for the object.

Returns the receiver if the requested lock was granted and was not dirty.

This method grants an exclusive lock on an object whenever it finds no reason to deny it. However, the lock that it grants may be dirty. One session's lock is dirty if another session has committed a change to the locked object since the beginning of the first session's current transaction. A session that holds a dirty lock cannot commit its transaction. To clean its locks, it must abort the transaction and obtain updated values for each object whose lock is dirty.

This method generates an error if the requested lock is denied. It also generates an error if the lock that it grants is dirty, but in this case the lock remains, even after the transaction is aborted.

`exclusiveLock:` *anObject* `ifDenied:` *denyBlock* `ifChanged:` *changeBlock*

Requests an exclusive lock on *anObject*. This method denies an exclusive lock on *anObject* under any one of the following circumstances:

- Another session already holds any kind of lock to the object.
- You do not have write authorization for the object.
- The object is special.

Returns the receiver if the requested lock was granted and was not dirty.

This method grants an exclusive lock on *anObject* whenever it finds no reason to deny it. However, the lock that it grants may be dirty. One session's lock is dirty if another session has committed a change to the locked object since the beginning of the first session's current transaction. A session that holds a dirty lock cannot commit its transaction. To clean its locks, it must abort the transaction and obtain updated values for each object whose lock is dirty.

This method generates an error if you do not have write authorization for *anObject*. If the requested lock is otherwise denied, it returns the value of the zero-argument block *denyBlock*. If it grants a dirty lock, then it returns the value of the zero-argument block *changeBlock*. In that case the lock remains, even after the transaction is aborted.

`exclusiveLockAll:` *aCollection*

Requests an exclusive lock on each object in *aCollection*. This method denies an exclusive lock on an object under the following circumstances:

- Another session already holds any kind of lock to the object.
- You do not have write authorization for the object.

If you lack write authorization for any object in *aCollection*, this method generates an error and no locks are granted. Otherwise, this method acquires locks on as many objects in *aCollection* as possible.

This method grants an exclusive lock on an object whenever it finds no reason to deny it. However, a lock that it grants may be dirty. One session's lock is dirty if another session has committed a change to the locked object since the beginning of the first session's current transaction. A session that holds a dirty lock cannot commit its transaction. To clean its locks, it must abort the transaction and obtain updated values for each object whose lock is dirty.

If a lock was acquired for every element of *aCollection*, and no locks are dirty, returns the receiver.

This method generates an error if it is unable to acquire a lock for every element of *aCollection*, or if any lock that it acquires is dirty. However, all the locks that it acquires remain in place, even after the current transaction is aborted.

`exclusiveLockAll:` *aCollection* `ifIncomplete:` *incompleteBlock*

Requests an exclusive lock on each object in *aCollection*. This method denies an exclusive lock on an object under any one of the following circumstances:

- Another session already holds any kind of lock to anObject.
- You do not have write authorization for anObject.
- The object is special.

If you lack write authorization for any object in *aCollection*, this method generates an error and no locks are granted. Otherwise, this method acquires locks on as many objects in *aCollection* as possible. If all requested locks were granted and none of the locks are dirty, returns the receiver.

This method grants an exclusive lock on an object whenever it finds no reason to deny it. However, a lock that it grants may be dirty. One session's lock is dirty if another session has committed a change to the locked object since the beginning of the first session's current transaction. A session that holds a dirty lock cannot commit its transaction. To clean its locks, it must abort the transaction and obtain updated values for each object whose lock is dirty.

If this method is unable to acquire a lock for every element of *aCollection*, or if any lock that it acquires is dirty, then it returns the value of the three-argument block *incompleteBlock*. The arguments to the block are:

1. An Array of objects that could not be locked.
2. An Array of objects that were locked but whose locks are dirty.
3. An empty Array, retained for backward compatibility with GemStone version 3.2. It was used formerly to hold uncommitted objects, which could not then be locked.

All the locks that it acquires remain in place, even after the current transaction is aborted.

`readLock:` *anObject*

Analogous to `System | exclusiveLock:ifDenied:ifChanged:`, but these methods differ in these respects:

- This method requests and grants read locks.
- This method grants a read lock on *anObject* if another session already holds a read lock, but grants no lock if another session already holds an exclusive or a write lock to *anObject*.
- This method requires only read authorization for the object, not write.

`readLock:` *anObject* `ifDenied:` *denyBlock* `ifChanged:` *changeBlock*

Analogous to `System | exclusiveLock:ifDenied:ifChanged:`, but these methods differ in these respects:

- This method requests and grants read locks.
- This method grants a read lock on *anObject* if another session already holds a read lock, but grants no lock if another session already holds an exclusive or a write lock to *anObject*.
- This method requires only read authorization for the object, not write.

`readLockAll:` *aCollection*

Analogous to `System | exclusiveLockAll:`, but these methods differ in these respects:

- This method requests and grants read locks.
- This method denies a read lock to the object if another session already holds an exclusive or a write lock to it. However, it grants a read lock if another session already holds a read lock.
- This method requires only read authorization for the object, not write.

`readLockAll:` *aCollection* `ifIncomplete:` *incompleteBlock*

Analogous to `System | exclusiveLockAll:ifIncomplete:`, but these methods differ in these respects:

- This method requests and grants read locks.
- This method denies a read lock to the object if another session already holds an exclusive or a write lock to it. However, it grants a read lock if another session already holds a read lock.
- This method requires only read authorization for the object, not write.

`writeLock:` *anObject* Analogous to `System | exclusiveLock:`. However, this method requests and grants write locks.

`writeLock:` *anObject* `ifDenied:` *denyBlock* `ifChanged:` *changeBlock*

Analogous to `System | exclusiveLock:ifDenied:ifChanged:`. However, this method requests and grants write locks.

`writeLockAll:` *aCollection*

Analogous to `System | exclusiveLockAll:`. However, this method requests and grants write locks.

`writeLockAll:` *aCollection* `ifIncomplete:` *incompleteBlock*

Analogous to `System | exclusiveLockAll:ifIncomplete:`. However, this method requests and grants write locks.

Signals

`sendSignal:` *aSignal* `to:` *aSessionId* `withMessage:` *aString*

Sends a signal (a SmallInteger) to the specified session (a SmallInteger) with *aString* as a message. The *aString* argument is currently limited to 1023 bytes.

System Control

`activeRepositories` Returns an Array containing references to the repositories that are attached at the time the message is sent.

`addAllToStoneLog:` *aString*

Appends text to the stone's informational log file. First, this method writes a banner that identifies the session from which *aString* came. It then appends *aString* itself. The argument must be a kind of String or DoubleByteString.

`concurrencyMode` Returns the Concurrency Mode in the form of one of the following symbols: #FULL_CHECKS or #NO_RW_CHECKS.

`concurrencyMode:` *aString*

Sets the Concurrency Control Mode, where *aString* is one of the following symbols: #FULL_CHECKS or #NO_RW_CHECKS.

The default is #FULL_CHECKS, which enables read-write and write-write conflict checking. In #NO_RW_CHECKS, a read-write conflict won't cause commit failure, but a read set is still maintained to support object locking.

If you are not the only user logged in to GemStone this method generates an error. To execute this method, you must have the SessionAccess privilege.

The Garbage Collector session is shut down for the duration of this method.

`resumeLogins` Allows new sessions to be initiated. (Enables users to login.) Logins are enabled when the GemStone system is started. This message reverses the effect of `System | suspendLogins`. Requires the SystemControl privilege.

`shutDown` Aborts all current sessions, then terminates them. Finally, the GemStone system is shut down. The session issuing this message terminates with a broken connection. Requires the SystemControl privilege.

`suspendLogins` Prevents any new sessions from being initiated. That is, no new user is allowed to login. However, users already active will be allowed to continue processing.

To reenable logins, send the message `System | resumeLogins`. (If you fail to do so, GemStone automatically reenables logins when the last user logs out.)

Requires the SystemControl privilege.

Time

`timeGmt` Returns a LargePositiveInteger, the time since January 1, 1970, in seconds. The time is computed from the clock of the machine on which the session is running, using the offset from the clock on the stone's (GemStone repository monitor process) machine which is cached in the session at login.

`timeGmt95` Returns a SmallInteger, the time since January 1, 1995, in seconds. The time is computed from the clock of the machine on which the session is running, using the offset from the clock on the stone's (GemStone repository monitor process) machine which is cached in the session at login.

Transaction Control

Transactions are discussed in detail in the *GemStone Programming Guide*.

`abortTransaction` Rolls back all modifications made to committed GemStone objects and provides the session with a new view of the most recently committed GemStone state.

These operations are performed whether or not the session was previously in a transaction. If the transaction mode is set to #autoBegin, then a new transaction is started. If the transaction mode is set to manualBegin, then a new transaction is not started.

`beginTransaction` Starts a new transaction for the session. If the session is already in a transaction, aborts the transaction and starts a new transaction.

If any permanent objects had been written by the session, their state is aborted. This method returns the receiver (System).

`checkpoint`

Attempt to commit the transaction for the current session, as a checkpoint. If full logging mode is in use, this method waits for any asynchronous checkpoints already in progress to complete, and then starts an asynchronous checkpoint. If partial logging mode is in use, this method performs a synchronous checkpoint.

This method is the same as `commitAndReleaseLocks` except for the checkpoint. If the current transaction is a read-only transaction, then this method is equivalent to `commitTransaction`, and a checkpoint is not written.

Frequent use of this method will seriously degrade performance. If partial logging mode is in use and extents are replicated, it can be used to ensure that a large transaction is recoverable after a disk failure. Otherwise, it should only be used for quality assurance testing.

`commitAndReleaseLocks`

Attempt to commit the transaction for the current session.

This method is the same as `commitTransaction` except for the handling of locks. If the commit succeeds, this method releases all locks for the session and returns true. Otherwise, it returns false and does not release locks.

This method also clears the commit release locks and commit-or-abort release locks sets. See the 'Releasing Locks' method category for more information.

`commitTransaction`

Attempts to update the persistent state of the Repository to include changes made by this transaction.

If the commit operation succeeds, then this method returns true, and the current transaction's changes, if any, become a part of the persistent Repository. After the repository update, the session exits the current transaction. If the transaction mode is autoBegin, then the session enters a new transaction. If the transaction mode is manualBegin, then the session remains outside of a transaction.

If conflicts prevent the repository update, then this method returns false. Call the `transactionConflicts` method to determine the nature of the conflicts. If the session is outside of a transaction, then this method raises the error rtErrPrimOutsideTrans.

This method also updates the session's view of GemStone. If the commit operation succeeds, then all objects in the session's view are consistent with the current state of GemStone. If the commit fails, then this method retains all the changes that were made to objects within the current transaction. However, commits made by other sessions are visible to the extent that changes in this transaction do not conflict with them.

`continueTransaction` Updates the session's view to the most recently committed GemStone state without rolling back modifications made to committed GemStone objects. The read and write sets of the session are carried forward and continue to accumulate until the session either commits or aborts. Changes made by this session to committed objects are not visible to other sessions until the session commits.

Returns true if accumulated modifications to the committed objects would not cause concurrency conflict as of the new view; otherwise returns false. If it returns false, you can call the `transactionConflicts` method to determine the nature of the conflicts.

Warning:
Once `continueTransaction` *has been used within a transaction, a subsequent commit of that transaction will ignore read-write and write-read conflicts. To check for read-write and write-read conflicts, a transaction could use the sequence* `continueTransaction`, `transactionConflicts`, `commitTransaction` *and check the result of* `transactionConflicts` *before doing the* `commitTransaction`.

This method can be used whether or not the session is outside of a transaction. Of course, the session cannot commit the accumulated changes unless it is inside a transaction.

If transaction mode is manualBegin, then `continueTransaction` does not alter the inside/outside of transaction state of the session.

Modifications made by other committed transactions are accumulated for retrieval by GciDirtyObjs() and GciDirtySavedObjs() just as they are accumulated for `commitTransaction` or `abortTransaction`.

This method has no effect on object locks held by the session. Locks in the release locks sets are not released.

`disableSignaledAbortError`

Disables the generation of an error when stone signals the gem session that it should abort when running outside of a transaction.

`enableSignaledAbortError`

Enables the generation of an error when the stone has signaled that the gem process should abort to connect to a more current GemStone root.

This method must be invoked after each delivery of the signal-abort error, to renable generation of the error.

`inTransaction` Returns true to indicate that the session is in a transaction, false otherwise.

`signaledAbortErrorStatus`

Returns true to indicate that the system generates an error when it receives the abort signal from stone. (In other words, verify that `enableSignaledAbortError` has been called to activate detection of the RT_ERR_SIGNAL_ABORT signal.) Returns false otherwise.

`transactionConflicts` Returns a SymbolDictionary that contains an Association whose key is #commitResult and whose value is one of the following Symbols:

```
#success
#failure
#retryFailure
#commitDisallowed
#rcFailure
#allSymbolsFailure
```

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. If there are no conflicts for the transaction, the returned SymbolDictionary has no additional Associations.

The conflict sets are cleared at the beginning of a commit or abort and therefore may be examined until the next commit, continue or abort.

The keys for the conflicts are as follows:

Key	Conflicts
Read-Write	ReadSet and WriteSetUnion
Write-Read	WriteSet and ReadSetUnion
Write-Write	WriteSet and WriteSetUnion
Read-ExclusiveLock	ReadSet and ExclusiveLockSet
Write-ReadLock	WriteSet and ExclusiveLockSet
Write-WriteLock	WriteSet and WriteLockSet
Rc-Write-Write	Logical write-write conflict on reduced conflict object

Note:
You should be sure to disconnect conflict sets before committing to avoid making them persistent.

`transactionMode` Returns the current transaction mode for the current GemStone session, either #autoBegin or #manualBegin. The default is #autoBegin.

`transactionMode:` *newMode*

Sets a new transaction mode for the current GemStone session and exits the previous mode by aborting the current transaction. Valid arguments are #autoBegin and #manualBegin.

User-Defined Actions

`hasUserAction:` *aSymbol*

Returns true if the user action named *aSymbol* is installed in this GemStone session. Returns false otherwise.

`loadUserActionLibrary:` *aString*

Loads the session user action library specified by *aString*. This method always returns the receiver (System).

`systemUserActionReport`

Returns a SymbolDictionary that provides information about GemStone system user actions. These are user actions that are automatically installed in every GemStone session to support classes such as GsFile and GsSocket.

In the resulting SymbolDictionary, the keys are the symbolic names of the user actions, and the values are Booleans. A values is true if the user action is linked with your application, and false if the user action is linked with the current GemStone session.

`userAction:` *aSymbol* Invokes the user-defined action represented by *aSymbol*. Generates an error if the user action is not installed in this session, or if it expects any arguments.

A maximum of 47 user actions may be active at any one time on the current GemStone Smalltalk stack.

`userAction:` *aSymbol* `with:` *anArg*

Invokes the user-defined action represented by *aSymbol*, passing it the argument *anArg*. Generates an error if the user action is not installed in this session, or if the number of arguments expected by the user action is not 1.

A maximum of 47 user actions may be active at any one time on the current GemStone Smalltalk stack.

`userAction:` *aSymbol* `with:` *firstArg* `with:` *secondArg*

Invokes the user-defined action represented by *aSymbol*, passing it the arguments *firstArg* and *secondArg*. Generates an error if the user action is not installed in this session, or if the number of arguments expected by the user action is not 2.

A maximum of 47 user actions may be active at any one time on the current GemStone Smalltalk stack.

`userAction:` *aSymbol* `with:` *firstArg* `with:` *secondArg* `with:` *thirdArg*

Invokes the user-defined action represented by *aSymbol*, passing it the arguments *firstArg*, *secondArg*, and *thirdArg*. Generates an error if the user action is not installed in this session, or if the number of arguments expected by the user action is not 3.

A maximum of 47 user actions may be active at any one time on the current GemStone Smalltalk stack.

`userAction:` *aSymbol* `with:` *firstArg* `with:` *secondArg* `with:` *thirdArg* `with:` *fourthArg*

Invokes the user-defined action represented by *aSymbol*, passing it the arguments *firstArg*, *secondArg*, *thirdArg*, and *fourthArg*. Generates an error if the user action is not installed in this session, or if the number of arguments expected by the user action is not 4.

`userAction:` *aSymbol* `with:` *firstArg* `with:` *secondArg* `with:` *thirdArg* `with:` *fourthArg* `with:` *fifthArg*

Invokes the user-defined action represented by *aSymbol*, passing it the arguments *firstArg*, *secondArg*, *thirdArg*, *fourthArg*, and *fifthArg*. Generates an error if the user action is not installed in this session, or if the number of arguments expected by the user action is not 5.

`userAction:` *aSymbol* `with:` *firstArg* `with:` *secondArg* `with:` *thirdArg* `with:` *fourthArg* `with:` *fifthArg* `with:` *sixthArg*

Invokes the user-defined action represented by *aSymbol*, passing it the arguments *firstArg*, *secondArg*, *thirdArg*, *fourthArg*, *fifthArg*, and *sixthArg*. Generates an error if the user action is not installed in this session, or if the number of arguments expected by the user action is not 6.

`userAction:` *aSymbol* `with:` *firstArg* `with:` *secondArg* `with:` *thirdArg* `with:` *fourthArg* `with:` *fifthArg* `with:` *sixthArg* `with:` *seventhArg*

Invokes the user-defined action represented by *aSymbol*, passing it the arguments *firstArg*, *secondArg*, *thirdArg*, *fourthArg*, *fifthArg*, *sixthArg*, and *seventhArg*. Generates an error if the user action is not installed in this session, or if the number of arguments expected by the user action is not 7.

`userAction:` *aSymbol* `with:` *firstArg* `with:` *secondArg* `with:` *thirdArg* `with:` *fourthArg* `with:` *fifthArg* `with:` *sixthArg* `with:` *seventhArg* `with:` *eighthArg*

Invokes the user-defined action represented by *aSymbol*, passing it the arguments *firstArg*, *secondArg*, *thirdArg*, *fourthArg*, *fifthArg*, *sixthArg*, *seventhArg*, and *eighthArg*. Generates an error if the user action is not installed in this session, or if the number of arguments expected by the user action is not 8.

`userAction:` *aSymbol* `withArgs:` *anArray*

Invokes the user-defined action represented by *aSymbol*, passing it the elements of *anArray* as arguments. Generates an error if the user action is not installed in this session, or if the number of arguments expected by the user action is not the same as the number of elements in *anArray*.

A maximum of 47 user actions may be active at any one time on the current GemStone Smalltalk stack.

`userActionReport` Returns a SymbolDictionary that provides information about all user actions installed in this GemStone session. In that SymbolDictionary, the keys are the symbolic names of the user actions, and the values are Booleans (true if the user action is linked with your application, false if the user action is linked with the current GemStone session).

Version Management

`clientVersionAt:` *aSymbol*

Returns version information about the client GemBuilder for C process. If the client is a session using the linkable GemBuilder for C login, this method is equivalent to `gemVersionAt:`.

See `System(C) | gemVersionAt:` for further details.

`clientVersionReport` Return a SymbolDictionary whose keys are the names of operating system, hardware, or GemStone version attributes, and whose values are the current values of those attributes in the client GemBuilder for C process.

`gemVersionAt:` *aSymbol* Returns information about the gem process of the current session. *aSymbol* must be equal to a key in VersionParameterDict, otherwise nil is returned. The semantics of these keys are:

aSymbol	meaning
#cpuKind	detailed CPU type obtained at runtime: sun4m, '486'.
#cpuArchitecture	target CPU for which GemStone was compiled: SPARC, 'X86'.
#gsBuildArchitecture	operating system name and CPU for which GemStone was compiled: 'SunOs SPARC', 'NT Intel'.
#gsBuildDate	time at which the gem executable was compiled (a String).
#gsRelease	major and minor version of GemStone, such as '5.0.0'.
#gsVersion	major version of GemStone, such as '5.0'.
#imageKind	a Symbol: `#server`.
#nodeName	network node name: speedy.
#osName	operating system name: SunOs, NT.
#osRelease	release number of the operating system: '4.1.3', '3.5'.
#osVersion	vendor defined major version of the OS: '3'.

#processId	operating system process identifer (an Integer): 13529.
#processorCount	number of processors on the machine running the process.

`gemVersionReport` Return a SymbolDictionary whose keys are the names of operating system, hardware, or GemStone version attributes, and whose values are the current values of those attributes in the gem process.

`imageVersionAt:` *aSymbol*

Returns information about the GemStone kernel class image where *aSymbol* is one of the following:

#gsBuildDate	DateTime of last kernel class filein or upgrade.
#gsRelease	Version String of last kernel class filein or upgrade: '5.0.0'.
#gsVersion	Major version of image, such as '5.0' .
#imageKind	A Symbol: `#server`.

If *aSymbol* is not one of the above, returns nil.

`stoneVersionAt:` *aSymbol*

Returns version information about the stone (repository monitor) process.

See `System (C) | gemVersionAt:` for further details. *aSymbol* = #imageKind returns nil for the stone.

`stoneVersionReport` Return a SymbolDictionary whose keys are the names of operating system, hardware, or GemStone version attributes, and whose values are the current values of those attributes in the gem process.

Time

An instance of Time describes a time of day with one-second resolution. The class Time also provides methods for examining the system clock and for measuring the performance of a block.

The internal representation of a Time is based on Greenwich Mean Time. However, many methods express time in the local timezone. ("Local" time is local to your Gem process.) These methods automatically convert between timezones, but the internal representation remains in Greenwich Mean Time. Hence, you can interact with Time methods in a natural way, but Time objects can be safely compared to each other no matter what time zone is used to express them.

You can convert a Time to a String (using Formatting instance methods), and you can convert a String to a Time (using Instance Creation class methods). Such conversions require a specification to describe the format of the String. Some methods provide for the default format, HH:MM:SS, which uses a 24-hour clock.

Explicit string-formatting specifications take the form of an Array, described in Table 2.3. A specification is incorrect if it is missing an element or if an element value is not one of the acceptable values listed in the table.

Table 2.3 String-formatting Specification Array for Time

Element	Acceptable Value	Explanation
1st	A Character literal (such as $: or $.)	Separates hours, minutes, and seconds.
2nd	true	Include seconds.
2nd	false	Do not include seconds.
3rd	true	Time is expressed in 12-hour format, with am or pm (such as 1:30:55 pm). The space is required preceding the am or pm indicator.
3rd	false	Time is expressed in 24-hour format (such as 13:30:55).

Superclasses Magnitude, Object

Named Instance Variables **seconds —** The number of seconds since midnight, Greenwich Mean Time.

Instance Format Pointer, Nonindexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`at:` *anIndex* `put:` *aValue* Disallowed. You may not change the value of a Time.

`hours` Returns a SmallInteger (between zero and 23 inclusive) that gives the number of hours represented by the receiver since midnight, local time.

`hoursGmt` Returns a SmallInteger (between zero and 23 inclusive) that gives the number of hours represented by the receiver since midnight, Greenwich Mean Time.

`minutes` Returns a SmallInteger (between zero and 59 inclusive) that gives the number of minutes represented by the receiver since the previous hour, local time.

`minutesGmt` Returns a SmallInteger (between zero and 59 inclusive) that gives the number of minutes represented by the receiver since the previous hour, Greenwich Mean Time.

Arithmetic

`addSeconds:` *anInteger* Returns a Time that describes a time of day *anInteger* seconds later than that of the receiver.

`addTime:` *timeAmount* Returns a Time that describes a time of day that is *timeAmount* later than that of the receiver. The *timeAmount* argument can be an instance of Time, Date or DateTime.

`subtractSeconds:` *anInteger*
Returns a Time that describes a time of day *anInteger* seconds earlier than that of the receiver.

`subtractTime:` *timeAmount*
Returns a Time that describes a time of day that is *timeAmount* earlier than that of the receiver. The *timeAmount* argument can be an instance of Time, Date or DateTime.

Comparing

`<` *aTime*	Returns true if the receiver represents a time of day before that of the argument, and false if it doesn't. Generates an error if the argument is not a Time.
`=` *aTime*	Returns true if the receiver represents the same time of day as that of the argument, and false if it doesn't.
`>` *aTime*	Returns true if the receiver represents a time of day after that of the argument, and false if it doesn't. Generates an error if the argument is not a Time.
`hash`	Returns an Integer hash code for the receiver.

Converting

`asSeconds`	Returns an Integer that represents the receiver in units of seconds since midnight, Greenwich Mean Time.
`timeAsSeconds`	Returns a SmallInteger (between zero and 86399 inclusive) that gives the number of seconds represented by the receiver since midnight, Greenwich Mean Time.

Formatting

`asString`	Returns a String that expresses the receiver in local time in the default format (HH:MM:SS).
`asStringGmt`	Returns a String that expresses the receiver in Greenwich Mean Time in the default format (HH:MM:SS).
`asStringGmtUsingFormat:` *anArray*	Returns a String that expresses the receiver in Greenwich Mean Time in the format defined by *anArray*. Generates an error if *anArray* contains an incorrect formatting specification. See Table 2.3 for a complete description of the String-formatting specification Array.
`asStringUsingFormat:` *anArray*	Returns a String that expresses the receiver in local time in the format defined by *anArray*. Generates an error if *anArray* contains an incorrect formatting specification. See Table 2.3 for a complete description of the String-formatting specification Array.

`printOn:` *aStream* — Puts a displayable representation of the receiver, expressed in local time, on *aStream*.

Storing and Loading

`writeTo:` *passiveObj* — Writes the passive form of the receiver into *passiveObj*, expressed in Greenwich Mean Time.

Class Protocol

Adjusting

`gmtOffsetSeconds` — Returns a SmallInteger that gives the offset in seconds of the local time zone, its difference with respect to Greenwich Mean Time. The local time zone is the time zone of the machine where the current Gem process is running.

A positive number corresponds to west of Greenwich, a negative number to east of Greenwich. For example, the offset for the Pacific Standard Time zone is 28800.

`gmtOffsetSeconds:` *aSmallInteger*

Sets the offset in seconds of the local time zone, with respect to Greenwich Mean Time that is used by this session. This overrides the default, which is the time zone of the machine where the current Gem process is running. Returns the receiver.

A positive number corresponds to west of Greenwich, a negative number to east of Greenwich. For example, the offset for the Pacific Standard Time zone is 28800.

Instance Creation

`fromSeconds:` *anInteger* Creates and returns an instance of the receiver from the specified value, which expresses Greenwich Mean Time.

`fromStream:` *aStream* Creates and returns an instance of the receiver by reading a String from *aStream*. The String expresses local time in the default format (HH:MM:SS). Generates an error if the String does not conform to the format.

`fromStream:` *aStream* `usingFormat:` *anArray*

Creates and returns an instance of the receiver by reading a String from *aStream*. The String expresses local time in the format specified by *anArray*. The expression is terminated either by a space character or by the end of the Stream. Generates an error if the String does not conform to the format, or if *anArray* contains an incorrect formatting specification.

See Table 2.3 for a complete description of the String-formatting specification Array.

`fromStreamGmt:` *aStream*

Creates and returns an instance of the receiver by reading a String from *aStream*. The String expresses Greenwich Mean Time in the default format (HH:MM:SS). Generates an error if the String does not conform to the format.

`fromStreamGmt:` *aStream* `usingFormat:` *anArray*

Creates and returns an instance of the receiver by reading a String from *aStream*. The String expresses Greenwich Mean Time in the format specified by *anArray*. The expression is terminated either by a space character or by the end of the Stream. Generates an error if the String does not conform to the format, or if *anArray* contains an incorrect formatting specification.

See Table 2.3 for a complete description of the String-formatting specification Array.

`fromString:` *aString* Creates and returns an instance of the receiver from the String *aString*. The String expresses local time in the default format (HH:MM:SS). Generates an error if the String does not conform to the format.

`fromString:` *aString* `usingFormat:` *anArray*

Creates and returns an instance of the receiver from the String *aString*. The String expresses local time in the format specified by *anArray*. The expression is terminated either by a space character or by the end of the String. Generates an error if the String does not conform to the format, or if *anArray* contains an incorrect formatting specification.

See Table 2.3 for a complete description of the String-formatting specification Array.

`fromStringGmt:` *aString* Creates and returns an instance of the receiver from the String *aString*. The String expresses Greenwich Mean Time in the default format (HH:MM:SS). Generates an error if the String does not conform to the format.

`fromStringGmt:` *aString* `usingFormat:` *anArray*

Creates and returns an instance of the receiver from the String *aString*. The String expresses Greenwich Mean Time in the format specified by *anArray*. The expression is terminated either by a space character or by the end of the String. Generates an error if the String does not conform to the format, or if *anArray* contains an incorrect formatting specification.

See Table 2.3 for a complete description of the String-formatting specification Array.

`new` Disallowed. To create a new Time, use another instance creation method.

`new:` *anInteger* Disallowed. To create a new Time, use another instance creation method.

`now` Creates and returns an instance of the receiver from the system clock on the machine that is running the Gem process, which is assumed to represent the current time of day.

Measuring

`millisecondClockValue`

Returns a SmallInteger representing the current relative time in milliSeconds. The result is a SmallInteger between 0 and 524287999, equivalent to

```
(System _timeGmtFloat * 1000) asInteger
   \\ 524288000
```

The result is computed locally in the session process, using the offset from the Gem's time that was cached in the session at login.

`millisecondsElapsedTime:` *aBlock*

Returns the elapsed time in milliseconds *aBlock* takes to return its value. The argument *aBlock* must be a zero-argument block.

Storing and Loading

`loadFrom:` *passiveObj*

Creates and returns an active instance of the receiver from the passive form of the object, which expresses itself in Greenwich Mean Time.

UndefinedObject

This class describes the behavior of nil, the 'nonexistent' object. You may not create new instances of UndefinedObject.

Superclasses Object

Named Instance Variables None

Instance Format Special, Nonindexable, Invariant

Subclass Creation Disallowed

Instance Protocol

Clustering

`clusterDepthFirst` Returns true. (Because nil is a self-defining object, this method has no effect.)

Copying

`copy` Returns the receiver. The pseudovariable nil is the only instance of UndefinedObject, and must preserve identity.

Formatting

`asString` Returns the string nil.

`describeClassName` Returns a String to describe classes with nil names.

Storing and Loading

`writeTo:` *passiveObj* Converts the receiver to its passive form and writes that information on *passiveObj*.

Class Protocol

Instance Creation

`fromStream:` *aStream*	If the next characters in *aStream* are nil (case-insensitive), returns nil. Otherwise, generates an error.
`fromString:` *aString*	If the first three characters in *aString* are nil (case-insensitive), returns nil. Otherwise, generates an error.
`new`	Disallowed. You cannot create new instances of UndefinedObject.

Storing and Loading

`loadFrom:` *passiveObj*	Reads from *passiveObj* the passive form of an object. Converts the object to its active form by loading the information into a new instance of the receiver. Returns the new instance.

UnorderedCollection

UnorderedCollection is an abstract class for collections of objects whose elements are not logically arranged in any particular order. The elements are also not physically stored in any fixed order. Any implied ordering at any given time is independent of the order in which the elements were added to the collection and cannot be relied upon to persist.

The elements of unordered collections are all of the same kind. Unless restricted further by a subclass, the kind of elements in unordered collections is Object. That is, the class of each element must simply be some kind of Object.

You cannot add nil to any kind of unordered collection. Attempts to do so have no effect.

UnorderedCollection provides for fast associative access of collection elements in searches by means of the use of indexes with selection blocks. UnorderedCollection creates each index for an individual instance, where specified, and maintains that index thereafter unless it is removed explicitly.

Indexing is done on instance variables, not on values returned by messages. When an index path is used as an argument to a method, it is specified by a String that consists of instance variable names separated by periods (such as the String 'instvar1.instvar2.instvar3'). The ith name in the String corresponds to the ith position in the path. A path String may include up to 16 names and is limited to a total of 1024 characters.

If aPathString is an empty path (that is, a zero-length String), the method operates upon the elements of the receiver itself rather than upon the instance variables of those elements.

For more information about index structures and path expressions, see the *GemStone Programming Guide*.

Superclasses	Collection, Object
Named Instance Variables	None
Instance Format	Nsc, Nonindexable, Variant
Subclass Creation	Allowed

Instance Protocol

Accessing Indexes

`equalityIndexedPaths` Returns an Array of Strings, each of which represents a path for which an equality index exists in the receiver. Each path originates with the elements of the receiver.

`equalityIndexedPathsAndConstraints`
Returns an Array containing info about equality indexes. The array consists of String/Class pairs. The string represents a path of the receiver's element kind for which an equality index exists in the receiver. The class is the constraint on the last element in the path.

`identityIndexedPaths` Returns an Array of Strings, each of which represents a path for which an identity index exists in the receiver. Each path originates with the elements of the receiver.

`kindsOfIndexOn:` *aPathString*
Returns a Symbol that indicates the kinds of indexes into the receiver that exist on *aPathString*: #identity, #equality, #equalityAndIdentity, or #none (either *aPathString* is not a path for the element kind of the receiver, or no indexes into the receiver exist on *aPathString*).

Adding

`add:` *anObject* Includes *anObject* as an element of the receiver anInteger number of times. Generates an error if *anObject* is not a kind of the bag's element kind.

`add:` *anObject* `withOccurrences:` *anInteger*
Includes *anObject* as an element of the receiver *anInteger* number of times. Generates an error if *anObject* is not a kind of the bag's element kind.

Clustering

`clusterDepthFirst` Clusters the receiver and its named and unnamed instance variables in depth-first order. If indexes are present, the internal indexing objects are clustered also. Returns true if the receiver has already been clustered during the current transaction; returns false otherwise.

`clusterIndexes` Clusters internal indexing objects using the current default ClusterBucket.

Copying

`copy` Returns a copy of the receiver that shares the receiver's public instance variables but has no indexes.

Indexing Support

`getLastElementConstraintOnPath:` *aPathString*

Returns the class that is the last constraint class along the given path string. If any of the classes that are traversed is not constrained on the given instance variable name, nil is returned.

Removing

`removeAllPresent:` *aCollection*

Removes from the receiver one occurrence of each element of *aCollection* that is also an element of the receiver. Differs from removeAll: in that, if some elements of *aCollection* are not present in the receiver, no error is generated. Returns the receiver.

`removeIfPresent:` *anObject*

Removes *anObject* from the receiver and returns the receiver. If *anObject* is present several times in the receiver, only one occurrence is removed. Returns nil if *anObject* is missing from the receiver.

Searching

`detect:` *aBlock* Evaluates *aBlock* repeatedly, with elements of the receiver as the argument. Returns the first element for which *aBlock* evaluates to true. If none of the receiver's elements evaluates to true, generates an error. The argument *aBlock* must be a one-argument block. Uses associative access when the argument is a SelectionBlock.

`detect:` *aBlock* `ifNone:` *exceptionBlock*
Evaluates *aBlock* repeatedly, with elements of the receiver as the argument. Returns the first element for which *aBlock* has the value true. If none of the receiver's elements has the value true, this evaluates the argument *exceptionBlock* and returns its value. The argument *aBlock* must be a one-argument block, and *exceptionBlock* must be a zero-argument block. Uses associative access when the argument is a SelectionBlock.

`includes:` *anObject*
Returns true if *anObject* is equal to one of the elements of the receiver. Returns false otherwise.

`includesIdentical:` *anObject*
Returns true if *anObject* is identical to one of the elements of the receiver. Returns false otherwise.

`includesValue:` *anObject*
Returns true if the receiver contains an object of the same value as the argument, *anObject*. Returns false otherwise. (Compare with `includes:`, which is based on identity.)

`occurrencesOf:` *anObject*
Returns the number of the receiver's elements that are identical (==) to *anObject*.

`reject:` *aBlock*
Evaluates *aBlock* with each of the receiver's elements as the argument. Stores the values for which *aBlock* is false into a collection of the same class as the receiver, and returns the new collection. The argument *aBlock* must be a one-argument block. Uses associative access when the argument is SelectionBlock.

`select:` *aBlock*
Evaluates *aBlock* with each of the receiver's elements as the argument. Stores the values for which *aBlock* is true into a collection of the same class as the receiver, and returns the new collection. The argument *aBlock* must be a one-argument block. Uses associative access when the argument is a SelectionBlock.

The new collection returned by this method will not retain any indexes of the receiver. If you want to perform indexed selections on the new collection, you must build all of the necessary indexes. The discussion of 'Transferring Indexes' in the 'Indexed Associative Access' chapter of the *GemStone Programming Guide* describes a technique for doing this.

`selectAsStream:` *aBlock*

Same functionality as `select:` except that the result is returned as a RangeIndexReadStream rather than an IdentitySet. The select block is limited in the following ways:

- The select block may only contain a single predicate.
- The predicate must contain one path expression.
- An equality index must exist for the path expression.

To use the stream that this method returns most effectively, avoid modifying both the receiver of this message and the selected objects returned by the stream as long as the stream is being accessed. Changes that alter the equality index can cause stream access failures.

Storing and Loading

`basicLoadFrom:` *passiveObj* `size:` *varyingSize*

Reads from *passiveObj* the passive form of an object. Converts the object to its active form by loading the information into the receiver.

`basicLoadFromNoRead:` *passiveObj* `size:` *varyingSize*

Reads from *passiveObj* the passive form of an object. Converts the object to its active form by loading the information into the receiver.

`basicWriteTo:` *passiveObj*

Converts the receiver to its passive form and writes that information on *passiveObj*.

`loadFrom:` *passiveObj* `size:` *varyingSize*

Reads from *passiveObj* the passive form of an object. Converts the object to its active form by loading the information into the receiver.

`loadNamedIVsFrom:` *passiveObj*

Reads named instance variables from the given passive object. The first instance variable should already have been parsed and be available in the *passiveObj* argument.

`loadVaryingFrom:` *passiveObj* `size:` *varyingSize*

Reads the varying part of the receiver from the given passive object. Does not record the receiver as having been read. Does not read the receiver's named instvars, if any.

Updating Indexes

`createEqualityIndexOn:` *aPathString*

Creates an equality index on *aPathString*. Generates an error if *aPathString* is not a path for the element kind of the receiver or if any term of the path is not constrained.

`createEqualityIndexOn:` *aPathString* `commitInterval:` *anInteger*

Creates an equality index on *aPathString*. Generates an error if *aPathString* is not a path for the element kind of the receiver or if any term of the path is not constrained. While index creation is in progress, a transaction commit is performed after the given number of elements in the Nsc have been processed. This method is intended to speed up index creation for very large collections.

`createEqualityIndexOn:` *aPathString* `withLastElementClass:` *aClass*

Creates an equality index on the path specified by *aPathString*. The equality index will be ordered according to the sort provided comparison operators provided by *aClass*.

`createEqualityIndexOn:` *aPathString* `withLastElementClass:` *aClass*
`commitInterval:` *anInteger*

Creates an equality index on the path specified by *aPathString*. The equality index is ordered according to the sort-provided comparison operators provided by *aClass*. While index creation is in progress, a transaction commit is performed after the given number of elements in the Nsc have been processed. This method is intended to speed up index creation for very large collections.

`createIdentityIndexOn:` *aPathString*

Creates an identity index on *aPathString*. Generates an error if *aPathString* is not a path for the element kind of the receiver or if any term of the path except the last term is not constrained.

`createIdentityIndexOn:` *aPathString* `commitInterval:` *anInteger*

Creates an identity index on *aPathString*. Generates an error if *aPathString* is not a path for the element kind of the receiver or if any term of the path except the last term is not constrained. While index creation is in progress, a transaction commit is performed after the given number of elements in the Nsc have been processed. This method is intended to speed up index creation for very large collections.

`progressOfIndexCreation`

Returns a String that describes the progress of an index creation that is underway.

`recomputeIndexSegments`

Clears the segments set of each path term for all indexes, then traverses all elements in the receiver, adding the segment into the corresponding path term's segment set. This method will have a high probability of causing concurrency conflicts with other sessions that modify the receiver.

`removeAllIndexes`
Remove all indexes for the receiver. If the receiver contains implicit indexes (due to its participation as a set-valued instance variable in another NSC's index), this method returns an array of pairs. The first object in each pair is a root NSC that has an index, and the second object in the pair is a path string that causes the receiver to participate in an index. If all of the receiver's indexes can be removed, this method returns the receiver.

`removeEqualityIndexOn:` *aPathString*
If an equality index exists on *aPathString*, remove that index. If the path string is invalid or no index exists on the given path, an error is raised. If *aPathString* is an implicit index (due to the receiver's participation as a set-valued instance variable in some other Nsc's index), then this method returns the path string.

`removeIdentityIndexOn:` *aPathString*
If an identity index exists on *aPathString*, and *aPathString* is not a proper prefix of some indexed path, the the index is removed. If the path string is invalid or no index exists on the given path, an error is raised. If *aPathString* is an implicit index (due to the receiver's participation as a set-valued instance variable in some other Nsc's index), then this method returns the path string.

`removeIncompleteIndex`
If there is an incomplete index, clean it up.

Class Protocol

Accessing the Class Format

`firstPublicInstVar` Returns the index of the first user-visible instance variable defined in this class, regardless of whether or not this class actually has user-visible instance variables.

`hasPublicInstVars` Returns true if this class has user-visible instance variables defined, false if not.

Storing and Loading

`loadFrom:` *passiveObj* Reads from *passiveObj* the passive form of an object. Converts the object to its active form by loading the information into a new instance of the receiver. Returns the new instance.

UserProfile

Each instance of UserProfile contains a number of characteristics associated with a given system user. For more information, see the *GemStone System Administration Guide* and the *GemStone Programming Guide*.

Superclasses

Object

Class Variables

LanguageNames — Obsolete in Gemstone 5.0.

AnArray of Symbols that identify the character set and language used during compilation. Valid compiler language names were #ASCII and #JIS-EUC.

PrivilegeNames — An Array of Symbols that identify the privileges (that is, the levels of access to certain privileged system functions) that may be assigned to a UserProfile. See UserProfile | privileges for a list of privilege names and their associated privileged methods.

Named Instance Variables

encryptedPassword — An InvariantString derived from the password that the user supplies for identification purposes at login. Limited to 1024 characters.

This instance variable is obsolete after repository conversion from a GemStone 4.1 repository. It is not used in GemStone 5.0.

userId — A String that identifies the user to the system at login; limited to 1024 characters. Methods in this class enforce uniqueness by value of all userIds of userProfiles in AllUsers.

symbolList — An Array of SymbolDictionaries that are used for resolving compile-time symbols.

defaultSegment — The Segment in which new objects are stored after each login.

privileges — A SmallInteger describing the level of access to certain privileged system functions that are ordinarily performed by the GemStone data curator.

groups — A SymbolSet; the groups to which the user belongs.

spare1 — Reserved for future use.

spare2 — Reserved for future use.

spare3 — Reserved for future use.

compilerLanguage — Obsolete. Was a SmallInteger describing the current compiler language environment.

Instance Format Pointer, Nonindexable, Variant

Subclass Creation Disallowed

Instance Protocol

Accessing

`activeUserIdLimit` Returns the maximum number of concurrent logins allowed for the receiver.

Generates an error if you do not have OtherPassword privilege.

`defaultSegment` Returns the default login Segment associated with the receiver.

`groups` Returns a SymbolSet, the set of groups (Symbols) to which the user belongs.

`isDisabled` Returns true if logins for the receiver are disabled, false otherwise.

Generates an error if you do not have OtherPassword privilege.

`lastLoginTime` Returns a DateTime of the last login time of the receiver, or nil if no login time has been recorded. If no age limits are set on passwords, no login times are recorded. (See updating methods in UserProfileSet.)

Generates an error if you do not have OtherPassword privilege.

`lastPasswordChange` Returns a DateTime that specifies when the password was last changed.

Generates an error if you do not have OtherPassword privilege.

`loginsAllowedBeforeExpiration`

Returns the number of logins allowed before the receiver's password will expire. Zero or nil means unlimited logins. A positive number is the number remaining before the password will expire. A negative number means the account has expired. The internal value is reset to zero when the password is changed.

Generates an error if you do not have OtherPassword privilege.

`nativeLanguage` Returns a Symbol specifying the user's native language. This value is used by error routines and other human interface routines to provide a system that converses in the user's native language.

`passwordNeverExpires` Returns true if the receiver is an account that is immune from password expiration. The SystemUser, DataCurator, and GcUser accounts are immune.

`reasonForDisabledAccount`

Returns the String that contains the reason why the receiver's password has been automatically disabled or expired.

Generates an error if you do not have OtherPassword privilege.

`userId` Each user is associated with a unique user identifier (a Symbol). This message returns that Symbol associated with the receiver.

Accessing Privileges

`privileges` Returns an Array of Strings describing the receiver's privileges. Those privilege Strings specify the user's level of access to certain privileged system functions ordinarily performed by the GemStone data curator.

Table 2.4 shows the privileged methods that are associated with each privilege String. If a method is listed with more than one privilege, all such privileges are required to execute the method. For example, to send the message `System | stopOtherSessions`, you must have both SystemControl and SessionAccess privileges.

For more information about privileges, see the *GemStone Programming Guide*.

Table 2.4 Privileges and Privileged Methods

Privileges	Privileged Methods
DefaultSegment	`UserProfile | defaultSegment:`

FileControl	`Repository | abortRestore,`
	`addTransactionLog:replicate:size:,`
	`commitRestore,`
	`continueFullBackupTo:MBytes:,`
	`createExtent:,`
	`createExtent:withMaxSize:,`
	`createReplicateOf:named:,`
	`disposeReplicate:,`
	`fullBackupTo:,`
	`fullBackupTo:MBytes:,`
	`restoreFromArchiveLogs,`
	`restoreFromBackup:,`
	`restoreFromBackups:,`
	`restoreFromCurrentLogs,`
	`restoreFromLog:,`
	`restoreStatus,`
	`startNewLog,`
	`shrinkExtents,`
	`setArchiveLogDirectories:...`
	`replicatePrefix:,`
	`timeToRestoreTo:`

GarbageCollection	`Repository | auditWithLimit:,`
	`findDisconnectedObjects,`
	`markForCollection,`
	`markGcCandidates,`
	`pagesWithPercentFree:,`
	`repairWithLimit:,`
	`reclaimAll,`
	`scavengePagesWithPercentFree:`

Table 2.4 Privileges and Privileged Methods

Privileges	Privileged Methods

OtherPassword	`UserProfile | activeUserIdLimit,`
	`activeUserIdLimit:,`
	`clearOldPasswords,`
	`isDisabled,`
	`lastLoginTime,`
	`lastPasswordChange,`
	`loginsAllowedBeforeExpiration,`
	`loginsAllowedBeforeExpiration:,`
	`password:,`
	`reasonForDisabledAccount,`
	`reasonForDisabledAccount:,`
	`userId:`
	`UserProfileSet | findDisabledUsers,`
	`findProfilesWithAgingPassword`

SegmentCreation	`Segment | newInRepository:`

SegmentProtection	`Segment | group:authorization:,`
	`groupNo:group:authorization:,`
	`ownerAuthorization:,`
	`worldAuthorization:`

SessionAccess	`GsSession | serialOfSession:,`
	`sessionIdOfSerial:,`
	`sessionWithSerialNumber:`
	`System | concurrencyMode:,`
	`currentSessionNames,`
	`descriptionOfSession:,`
	`stopOtherSessions,`
	`userProfileForSession:`

Statistics	`System | stoneStatistics`

SystemControl	`GsSession | stop`
	`System | changeCacheSlotIoLimit:to:,`
	`resumeLogins,`
	`shutDown,`
	`stopOtherSessions,`
	`stopSession:,`
	`suspendLogins`

UserPassword	`UserProfile | oldPassword:newPassword:`

Table 2.4 Privileges and Privileged Methods

Privileges	Privileged Methods
(various)	`System | configurationAt:put:`

Accessing the Symbol List

`dictionaryAndSymbolOf:` *anObject*

Returns the symbol list Dictionary that names *anObject*, and also returns the name which that Dictionary associates with *anObject*. More precisely, this returns an Array containing two elements:

- The Dictionary in the user's symbol list that contains an Association whose value is *anObject*.
- The Symbol which is that Association's key.

The symbol list is searched in the same order that the compiler searches it. (For more information about symbol resolution, see the *GemStone Programming Guide*.) If *anObject* is not found in the symbol list, returns nil.

`objectNamed:` *aSymbol* Returns the first object in the receiver's symbol list that has the given name. If no object is found with the given name, this returns nil.

`resolveSymbol:` *aString* Searches the receiver's symbol list for an Association whose key is equal to *aString*, and returns that Association. If no such Association is found in the symbol list, this returns nil.

`symbolList` Whenever the compiler is invoked, tokens in the method's source are bound to either Symbols found in the source, or to SymbolAssociations found in one of a number of SymbolDictionary passed to the compiler. This method returns the Array of such SymbolDictionary that are associated with the receiver.

`symbolResolutionOf:` *aString*

Searches the receiver's symbol list for *aString*. If *aString* is found, returns a formatted String describing the position in the symbol list of the Dictionary defining *aString*, the name of that Dictionary, and *aString*.

Generates an error if *aString* is not defined for the receiver.

Backward Compatibility

Methods in this category are obsolete and are provided only for compatibility with earlier releases of GemStone. They will be removed in a future release.

`compilerLanguage` Obsolete in GemStone 5.0.

GemStone 5.0 ignores the language environment setting. Instead, the compiler uses the class of the sourceString to control character-set dependent processing during compilation.

`compilerLanguage:` *aString*

Obsolete in GemStone 5.0. Has no effect.

GemStone 5.0 ignores the language environment setting. Instead, the compiler uses the class of the sourceString to control character-set dependent processing during compilation.

`dictionaryNames` Obsolete in GemStone 5.0.

Clustering

`clusterDepthFirst` This method clusters the receiver's user id, password, and symbol list. Because the password and user id are assumed to be byte objects, and because the symbol list may contain shared elements, no further traversal of the objects is done. Returns true if the receiver has already been clustered during the current transaction; returns false otherwise.

Storing and Loading

`writeTo:` *aPassiveObject* Instances of UserProfile cannot be converted to passive form. This method writes nil to *aPassiveObject* and stops GemStone Smalltalk execution with a notifier.

Updating

`activeUserIdLimit:` *aPositiveInteger*

Sets the maximum number of concurrent logins for the receiver to be *aPositiveInteger*. This change will take effect after this session commits.

Generates an error if you do not have OtherPassword privilege.

`addGroup:` *aGroupString* Adds the user to the group represented by *aGroupString*, and returns the receiver. If the user already belongs to the group *aGroupString*, no action occurs.

If *aGroupString* does not already exist in the global object AllGroups, generates an error.

`clearOldPasswords` Clears the set of old passwords for the receiver, thus permitting reuse of some passwords that had been previously disallowed.

Generates an error if you do not have OtherPassword privilege.

`createDictionary:` *aSymbol*

Creates a new SymbolDictionary. The new Dictionary is created with a single SymbolAssociation, whose key is *aSymbol* and whose value is the new SymbolDictionary itself.

Also creates a SymbolAssociation in the receiver's UserGlobals dictionary, with *aSymbol* as the key and the new dictionary as its value.

Returns the new SymbolDictionary. Generates an error if *aSymbol* is already defined in the receiver's symbol list, or if *aSymbol* is not a Symbol.

`defaultSegment:` *aSegment*

Redefines the default login Segment associated with the receiver, and returns the receiver.

If the receiver is the UserProfile under which this session is logged in, this method requires the DefaultSegment privilege.

If the receiver is not the UserProfile under which this session is logged in, you must have write authorization for the Segment where the receiver resides. Exercise extreme caution when executing this method. If, at the time you commit your transaction, the receiver no longer had write authorization for *aSegment*, that user's GemStone session will be terminated and the user will be unable to log back in to GemStone.

`loginsAllowedBeforeExpiration:` *aPositiveInteger*

Sets the number of logins allowed using the receiver before the receiver's password will expire. Zero means unlimited logins. A positive number is the number of logins to be allowed before the current password will expire. The internal value is reset to zero when the password is next changed.

Generates an error if you do not have OtherPassword privilege.

`nativeLanguage:` *aLanguageSymbol*

Redefines the user's native language to be *aLanguageSymbol*.

`oldPassword:` *firstString* `newPassword:` *secondString*

Modifies the receiver's password to be *secondString*. Generates an error if either argument is not a String, if *firstString* is not the receiver's password, or if the receiver is not the UserProfile of the current session. Generates an error if *secondString* is equivalent to the **userId** of the receiver. The new password (*secondString*) may not be longer than 1024 characters.

This method allows you to change your own password. To change the password of any other user, use `password:` instead.

This method requires the UserPassword privilege.

`password:` *aString* Modifies the receiver's password to be *aString*, and returns the receiver. If the argument is not a String, generates an error. Generates an error if *aString* is equivalent to the **userId** of the receiver. *aString* may not be longer than 1024 characters.

This method allows you to change the password of another GemStone user. To change your own password, use `oldPassword:newPassword:` instead.

This method requires the OtherPassword privilege.

`reasonForDisabledAccount:` *aString*

Updates the String that contains the reason why the receiver's password has been automatically disabled or expired.

Generates an error if you do not have OtherPassword privilege.

`removeGroup:` *aGroupString*

Removes the user from the group represented by *aGroupString*. If *aGroupString* is not a group defined in the global collection AllGroups, generates an error. If the user does not belong to the group, no action occurs.

`userId:` *newUserIdString* Modifies the **userId** associated with the receiver to be *newUserIdString*. *newUserIdString* must not be the **userId** of some other userProfile in AllUsers, or an error will be raised. Has no effect of *newUserIdString* is equal to the current **userId** of the receiver.

Requires write authorization to the Segment DataCuratorSegment, and requires OtherPassword privilege.

Updating Privileges

`addPrivilege:` *aPrivilegeString*

Adds *aPrivilegeString* to the receiver's **privileges**. Generates an error if *aPrivilegeString* is not a valid privilege name (as defined in the class variable PrivilegeNames).

`deletePrivilege:` *aPrivilegeString*

Removes *aPrivilegeString* from the receiver's **privileges**. Generates an error if *aPrivilegeString* is not a valid privilege name (as defined in the class variable PrivilegeNames).

`privileges:` *anArrayOfStrings*

Redefines the receiver's **privileges** to be those specified by *anArrayOfStrings*. Any privileges not contained in *anArrayOfStrings* will be turned off. Generates an error if any element of *anArrayOfStrings* is not a valid privilege name (as defined in the class variable PrivilegeNames).

Updating the Symbol List

`insertDictionary:` *aSymbolDictionary* `at:` *anIndex*

Inserts *aSymbolDictionary* into the receiver's symbol list. If the receiver is identical to 'GsSession currentSession userProfile', inserts *aSymbolDictionary* into the transient session symbol list as well. The insertion into the receiver's symbol list occurs first.

If *anIndex* is less than or equal to the size of the receiver's symbol list, inserts *aSymbolDictionary* into the symbol list at *anIndex*.

If *anIndex* is 1 greater than the size of the receiver's symbol list, appends *aSymbolDictionary* to the receiver's symbol list.

If *anIndex* is more than 1 greater than the size of the receiver's symbol list, or if *anIndex* is less than 1, generates an error.

If an error occurs as the result of the persistent insertion, no further action is taken and an error is generated. If the insertion completes correctly, *aSymbolDictionary* is inserted into the transient session symbol list.

If *anIndex* is 1, prepend *aSymbolDictionary* to the beginning of the transient symbol list.

Otherwise, finds the dictionary at (*anIndex* - 1) in the persistent symbol list, and searches for it by identity in the transient symbol list. If found, insert *aSymbolDictionary* just after it in the transient symbol list. If not found, append *aSymbolDictionary* to the end of the transient symbol list.

If an error occurs as a result of the insertion in the transient symbol list, the persistent symbol list is left in its new state, the transient symbol list is left in its old state and the error is silently ignored.

Note:
If the transient and persistent lists have different contents when an abort transaction occurs they will not be automatically synchronized after the abort. The persistent list will be rolled back to the committed state, but the transient list will not be rolled back.

`removeDictionaryAt:` *anIndex*

Removes the SymbolDictionary at position *anIndex* from the receiver's symbol list, thus decreasing the size of the receiver's symbol list by one, and, if the receiver is identical to 'GsSession currentSession userProfile', removes the SymbolDictionary at position *anIndex* from the transient symbol list's symbol list (subject to the constraints below). Returns the receiver's dictionaryNames String.

Generates an error if *anIndex* is not both a positive integer and less than the size of the receiver's symbol list. If an error occurs in removing from the receiver's symbol list, the transient symbol list is left alone, and the error is handled immediately.

If the removal from the receiver's symbol list succeeds, search in the transient symbol list for a SymbolDictionary identical to the one removed from the transient symbol list, and if found, remove it from the list. Removes only the first such element. If no such dictionary is found, silently return.

Note:
If the transient and persistent lists have different contents when an abort transaction occurs they will not be automatically synchronized after the abort. The persistent list will be rolled back to the committed state, but the transient list will not be rolled back.

`symbolList:` *aSymbolList*

Modifies the list of SymbolDictionaries associated with the receiver. If the receiver is identical to 'GsSession currentSession userProfile', replaces the contents of the session transient symbol list with the contents of *aSymbolList*. If an error occurs as a result of the modification to the persistent symbol list, the transient list is left unmodified and the error is handled immediately. If an error occurs as a result of the modification of the transient symbol list, the persistent symbol list is left in its new state, the transient symbol list is left in its old state, and the error is silently ignored.

Note:
If the transient and persistent lists have different contents when an abort transaction occurs they will not be automatically synchronized after the abort. The persistent list will be rolled back to the committed state, but the transient list will not be rolled back.

User Authorization

`validatePassword:` *aString*

This method allows an application to validate an operation that requires authentication by multiple individuals. Returns true if *aString* is the password of the receiver, returns false otherwise. Generates an error if the argument is not a String.

Class Protocol

Backward Compatibility

Methods in this category are obsolete and are provided only for compatibility with earlier releases of GemStone. They will be removed in a future release.

`newWithUserId:` *userIdString* `password:` *aString* `defaultSegment:` *aSegment* `privileges:` *anArrayOfStrings* `inGroups:` *aCollectionOfGroupStrings* `compilerLanguage:` *aLangString*

Obsolete in GemStone 5.0. Use the `newWithUserId:password:defaultSegment:privileges:inGroups:` method instead.

GemStone 5.0 ignores the language environment setting. Instead, the compiler uses the class of the sourceString to control character-set dependent processing during compilation.

Instance Creation

`new`

Disallowed. To create a new UserProfile, use `newWithUserId:...` instead.

`newWithUserId:` *userIdString* `password:` *passwordString* `defaultSegment:` *aSegment* `privileges:` *anArrayOfStrings* `inGroups:` *aCollectionOfGroupStrings*

Creates a new UserProfile with the associated characteristics. In so doing, creates a symbol list with three dictionaries: UserGlobals, Globals, and Published. The first Dictionary (UserGlobals) is created for the user's private symbols. Returns the new UserProfile.

Adds the new UserProfile to AllUsers. Creates the new UserProfile as a member of the group Subscribers and of the groups in *aCollectionOfGroupStrings*.

Before the user can login, the user must be authorized to read and write in the specified default Segment.

Generates an error if *passwordString* is equivalent to *userIdString* ignoring case (`equalsNoCase:`).

UserProfileSet

UserProfileSet is a concrete subclass of AbstractUserProfileSet that implements and enforces some account and password security features for all of its elements.

One instance of UserProfileSet, called AllUsers, is provided with a fresh GemStone server. All UserProfiles in GemStone belong to this set. AllUsers supports security features for all users. Only AllUsers affects GemStone accounts and logins; other instances of UserProfileSet do not. AllUsers also ensures uniqueness of userId Strings; no two UserProfiles can have the same Id.

GemStone, as shipped from the factory, disables all the security features supported by AllUsers. To activate any or all of those features, an administrator with the proper privileges must execute methods on AllUsers. Activated features can also be deactivated later by reapplying the settings that do not constrain GemStone's behavior.

Password format constraints are applied only after an administrator commits the changes in AllUsers. They are then enforced only when users change their own passwords with the UserProfile>>oldPassword:newPassword: method, not when administrators or other users make changes with methods that require the OtherPassword privilege. In addition, enforcement does not apply to existing passwords created before new constraints were committed.

Superclasses

AbstractUserProfileSet, IdentitySet, IdentityBag, UnorderedCollection, Collection, Object

Named Instance Variables

userIdDictionary — A StringKeyValueDictionary whose keys are userId Strings and whose values are the UserProfiles of the instance.

userSecurityData — The **userSecurityData** variable is used internally by GemStone.

passwordAgeLimit — The maximum Number of hours for which a user can retain a password. When a password is set, it expires this Number of hours later. The user must change the password before it expires, or else GemStone disables the account. Once a password has expired, an administrator must reset the password from another account before the user can login again.

Zero means there is no expiration time for passwords.

passwordAgeWarning — The maximum Number of hours prior to a password expiration time for which a user can login without a warning. If the user logs in to GemStone within this Number of hours before the password is due to expire, GemStone issues a warning about the impending expiration. This feature grants a user the opportunity to change the password conveniently and to prevent the account from being disabled.

staleAccountAgeLimit — The maximum Number of hours for which a user account can remain enabled without a login. Once the user logs in, he or she has up to this Number of hours to login again, or else GemStone disables the account. Once the account has been disabled, an administrator must reset the password from another account before the user can login again.

Zero means there is no expiration time for accounts.

disallowUsedPasswords — A Boolean. When set to true, GemStone does not permit a user to reuse any former password from that time forward. When set to false, GemStone permits users to reuse passwords as they wish.

disallowedPasswords — A Set of Strings that cannot be used as passwords. The userId Strings of GemStone users also cannot be used as passwords, even if they do not appear in this Set.

maxPasswordSize — A SmallInteger that gives maximum size of new passwords. Zero means there is no maximum.

minPasswordSize — A SmallInteger that gives minimum size of new passwords. Zero means there is no minimum.

maxRepeatingChars — A SmallInteger that gives maximum number of adjacent characters in a password that can have the same value. Zero means there is no maximum. The value 1 prevents passwords of the form aa, but not aba.

maxConsecutiveChars — A SmallInteger that gives maximum number of adjacent characters in a password that form an ascending or descending sequence of character values, such as "123" or "zyx". Zero means there is no maximum. Such sequences are determined by case-sensitive comparisons.

maxCharsOfSameType — A SmallInteger that gives maximum number of adjacent characters in a password that are permitted to have the same type (alphabetic, numeric, or special). Zero means there is no maximum.

Instance Format Nsc, Nonindexable, Variant

Subclass Creation Disallowed

Instance Protocol

Accessing

`disallowedPasswords` Returns the set of disallowed passwords defined for the receiver.

`disallowUsedPasswords`

Returns the value of the **disallowUsedPasswords** instance variable.

`maxCharsOfSameType` Returns the value of the **maxCharsOfSameType** instance variable.

`maxConsecutiveChars` Returns the value of the **maxConsecutiveChars** instance variable.

`maxPasswordSize` Returns the value of the **maxPasswordSize** instance variable.

`maxRepeatingChars` Returns the value of the **maxRepeatingChars** instance variable.

`membersOfGroup:` *aString*

Returns an IdentitySet containing the userId for each member of the specified group. If the group contains no members, returns an empty IdentitySet.

Generates an error if *aString* is not a kind of String, or if *aString* is not defined in the global object AllGroups.

`minPasswordSize` Returns the value of the **minPasswordSize** instance variable.

`passwordAgeLimit` Returns the value of the **passwordAgeLimit** instance variable.

`passwordAgeWarning` Returns the value of the **passwordAgeWarning** instance variable.

`staleAccountAgeLimit` Returns the value of the **staleAccountAgeLimit** instance variable.

`userWithId:` *aString* `ifAbsent:` *aBlock*

Searches the receiver for a UserProfile whose userId is equal to *aString*, and returns that UserProfile. Evaluates the argument *aBlock* if no userId is equal to *aString*.

Adding

If the receiver is not AllUsers, a new user will be unable to log in to GemStone. In addition, in order to log into GemStone, a user must be authorized to read and write in the default Segment that is specified for that user.

`add:` *aUserProfile* Adds a new UserProfile to the receiver. Generates an error if *aUserProfile* has a userId that duplicates an existing element of the receiver.

`addNewUserWithId:` *userIdString* `password:` *passwordString*

Creates a new UserProfile and adds it to the receiver. The new UserProfile has no privileges, and belongs to no groups. This method creates a new Segment, which is owned by the new user and assigned as the user's default segment. The new Segment is created with world-read permission.

This default method can be used by the data curator in batch user installations. Returns the new UserProfile.

If the receiver is not AllUsers, the new user will be unable to log in to GemStone.

`addNewUserWithId:` *userIdString* `password:` *passwordString*
`defaultSegment:` *aSegment* `privileges:` *anArrayOfStrings*
`inGroups:` *aCollectionOfGroupStrings*

Creates and returns a new UserProfile with the associated characteristics, and adds it to the receiver. Generates an error if the *userIdString* duplicates the userId of any existing element of the receiver.

`addNewUserWithId:` *userIdString* `password:` *passwordString*
`defaultSegment:` *aSegment* `privileges:` *anArrayOfStrings*
`inGroups:` *aCollectionOfGroupStrings* `compilerLanguage:` *aLangString*

Creates a new UserProfile with the associated characteristics and adds it to the receiver. Generates an error if the *userIdString* duplicates the userId of any existing element of the receiver. Returns the new UserProfile.

In order to log in to GemStone, the user must be authorized to read and write in the specified default Segment.

Disk Space Management

`findObjectsLargerThan:` *aSize* `limit:` *aLimit*

Searches the symbol list of each user in the receiver for named objects larger than *aSize*. Returns an Array of the form #[#[aUserId, aKey, anObject]] where aKey is the symbolic representation of anObject such that:

```
((AllUsers userWithId: aUserId)
          resolveSymbol: aKey) value
 == anObject
```

is true. For each user in the receiver, the search continues until there are no more named objects in the user's symbol list, or until the size of the result reaches the specified maximum *aLimit*.

Generates an error if an authorization violation occurs.

Formatting

`dictionaryNames` Returns a formatted String describing each user's symbol list. For each user, the String contains the userId, and the position and name of each Dictionary in that user's symbol list.

This method assumes that each Dictionary in the symbol list contains an Association whose value is that Dictionary. If any Dictionary does not contain such an Association, it is represented in the result String as '(unnamed Dictionary)'.

Group Membership

`addGroup:` *aGroupString* Adds all the users in the receiver to the group represented by *aGroupString*. If the current session does not have the authorizations required for this operation, raises an error.

`removeGroup:` *aGroupString*

Removes all the users in the receiver from the group represented by *aGroupString*. If the current session does not have the authorizations required for this operation, raises an error.

`usersInGroup:` *aGroupString*

Returns all the elements of the receiver that are in the group represented by *aGroupString*. If the current session does not have the authorizations required for this operation, raises an error.

Logging

`addMsgToSecurityLog:` *aString*

This method adds the date, time and userId prefix to the specified message string and includes the resulting string in the security log. In GemStone 5.0, the stone log file is the security log.

Querying

`findDisabledUsers`
Returns a SortedCollection of UserProfiles that are disabled. The result includes users whose accounts have been disabled because their passwords have expired, or whose accounts were not used within the interval defined by the **staleAccountAgeLimit**, or who failed to login within the number of tries specified by the stone configuration parameter.

Generates an error if you do not have OtherPassword privilege.

`findProfilesWithAgingPassword`
Returns a SortedCollection of UserProfiles whose passwords will expire sooner than **passwordAgeWarning** hours from now.

Generates an error if you do not have OtherPassword privilege.

Removing

`remove:` *anObject* `ifAbsent:` *aBlock*
Reimplemented to maintain the userId dictionary.

Updating

`disallowUsedPasswords:` *aBoolean*
Sets the value of the **disallowUsedPasswords** instance variable.

`maxCharsOfSameType:` *aPositiveInteger*
Sets the value of the **maxCharsOfSameType** instance variable.

`maxConsecutiveChars:` *aPositiveInteger*
Sets the value of the **maxConsecutiveChars** instance variable.

`maxPasswordSize:` *aPositiveInteger*
Sets the value of the **maxPasswordSize** instance variable.

`maxRepeatingChars:` *aPositiveInteger*
Sets the value of the **maxRepeatingChars** instance variable.

`minPasswordSize:` *aPositiveInteger*

Sets the value of the **minPasswordSize** instance variable.

`passwordAgeLimit:` *numberOfHours*

If *numberOfHours* is greater than zero, the passwords of all UserProfiles in the receiver other than those for SystemUser, DataCurator, and GcUser will expire the specified number of hours after they are last changed.

The argument *numberOfHours* must be a SmallInteger or a Float and must be at least zero and at most 536870911.

`passwordAgeWarning:` *numberOfHours*

If *numberOfHours* is greater than zero, warning of passwords about to expire will be given for logins that occur less than the specified number of hours before the password is to expire.

The argument *numberOfHours* must be a SmallInteger or a Float and must be at least zero and at most 536870911.

`staleAccountAgeLimit:` *numberOfHours*

If *numberOfHours* is greater than zero, the password for an acccount is disabled if the user does not login to the account at least as often as the specified number of hours. The users SystemUser, DataCurator, and GcUser are never disabled by this mechanism.

The argument *numberOfHours* must be a SmallInteger or a Float and must be at least zero and at most 536870911.

WriteStream

A WriteStream is a PositionableStream that allows its objects to be written, but not read.

Superclasses PositionableStream, Stream, Object

Named Instance Variables None

Instance Format Pointer, Nonindexable, Variant

Subclass Creation Allowed

Instance Protocol

Accessing

`contents` WriteStreams return the portion of their collection that has been written: the collection up to the next write-position.

`next` Disallowed. You cannot read a WriteStream.

Adding

`nextPut:` *anObject* Inserts *anObject* as the next element that the receiver can access for writing. Returns *anObject*.

`nextPutAll:` *aCollection* Inserts the elements of *aCollection* as the next elements that the receiver can access. Returns *aCollection*.

`nextPutAllBytes:` *aCharacterCollection*
Inserts the byte contents of *aCharacterCollection* as the next elements that the receiver can access. Returns aCollection. The receiver's collection must be a GsFile or a String.

Class Protocol

Instance Creation

`new` Disallowed. To create a new WriteStream, use the class method `on:` instead.

Method Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Index

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y