

---

GemStone®

***GemStone/S 64 Bit  
Topaz Programming  
Environment***

Version 3.0

June 2011

**vmware®**

GEMSTONE<sup>™</sup>  
.....S 64

---

## INTELLECTUAL PROPERTY OWNERSHIP

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

This documentation, or any part of it, may not be reproduced, displayed, photocopied, transmitted, or otherwise copied in any form or by any means now known or later developed, such as electronic, optical, or mechanical means, without express written authorization from VMware, Inc.

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

The software installed in accordance with this documentation is copyrighted and licensed by VMware, 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 the Commercial Software - Restricted Rights clause at 52.227-19 of the Federal Acquisitions Regulations (48 CFR 52.227-19) except that the government agency shall not have the right to disclose this software to support service contractors or their subcontractors without the prior written consent of VMware, Inc.

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

## COPYRIGHTS

This software product, its documentation, and its user interface © 1986-2011 VMware, Inc., and GemStone Systems, Inc. All rights reserved by VMware, Inc.

## PATENTS

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

## TRADEMARKS

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

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

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

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

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

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

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

**Linux** is a registered trademark of Linus Torvalds and others.

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

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

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

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

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

**VMware, Inc.**  
15220 NW Greenbrier Parkway  
Suite 150  
Beaverton, OR 97006

---

## About This Manual

This manual describes Topaz, the command-driven GemStone programming environment. You can use Topaz with the other GemStone development tools such as GemBuilder for C to build comprehensive database applications.

Topaz is especially useful for database administration tasks and batch-mode procedures. Because it is command driven and generates ASCII output on standard output channels, Topaz offers access to GemStone without requiring a window manager or additional language interfaces.

## Prerequisites

To make use of the information in this manual, you must be familiar with the GemStone/S 64 Bit system and the GemStone Smalltalk programming language. In addition, you should be familiar with your host operating system.

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

## How This Manual Is Organized

- Chapter 1, “Getting Started with Topaz,” introduces you to Topaz. You’ll learn how to run Topaz, how to log in to the GemStone server, how to create and execute GemStone Smalltalk code, and how to inspect GemStone objects.
- Chapter 2, “Debugging Your GemStone Smalltalk Code,” shows how to use Topaz to debug your GemStone Smalltalk code.
- Chapter 3, “Command Dictionary,” describes the Topaz commands in alphabetical order.
- Appendix A, “Topaz Command-Line Syntax,” lists the Topaz command line syntax.
- Appendix B, “Network Resource String Syntax,” lists the syntax for specifying the host machine for a GemStone file or process.

## Terminology Conventions

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

## Typographical Conventions

This document uses the following typographical conventions:

- Command lines you type are shown in **bold** type. For example:  

```
% env | grep GEM
```
- Topaz commands and UNIX commands and are shown in **bold** type. For example:  

```
copyextent
```
- Smalltalk methods and instance variables, file names and paths, and screen dialogue examples are shown in monospace type. For example:  

```
markForCollection
```

- Place holders that are meant to be replaced with real values are shown in *italic* type. For example:

*StoneName.conf*

## Other GemStone Documentation

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

- *Programming Guide for GemStone/S 64 Bit* – a programmer’s guide to GemStone Smalltalk, GemStone’s object-oriented programming language.
- *GemBuilder for Smalltalk Users’s Guide* – describes GemBuilder for Smalltalk, a programming interface that provides a rich set of features for building and running client Smalltalk applications that interact transparently with GemStone Smalltalk.
- *GemBuilder for C* – describes GemBuilder for C, a set of C functions that provide a bridge between your application’s C code and the application’s database controlled by GemStone.
- *System Administration Guide* – describes maintenance and administration of your GemStone/S system.

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

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

## Technical Support

### GemStone Website

**<http://support.gemstone.com>**

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

- **Documentation** for released versions of all GemStone products, in PDF form.
- **Downloads and Patches**, including past and current versions of GemBuilder for Smalltalk.
- **Bugnotes**, identifying performance issues or error conditions you should be aware of.
- **TechTips**, providing information and instructions that are not otherwise included in the documentation.
- **Compatibility matrices**, listing supported platforms for GemStone product versions.

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

### Help Requests

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

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

**Website:** <http://techsupport.gemstone.com>

**Email:** [techsupport@gemstone.com](mailto:techsupport@gemstone.com)

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

When submitting a request, please include the following information:

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

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

## **24x7 Emergency Technical Support**

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

## **Training and Consulting**

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

—  
|

---

<b><i>Chapter 1. Getting Started with Topaz</i></b>	<b>15</b>
1.1 Invoking Topaz . . . . .	16
1.2 Overview of a GemStone Session . . . . .	17
1.3 Remote Versus Linked Versions . . . . .	18
1.4 Logging In to GemStone . . . . .	18
Setting Up a Login Initialization File . . . . .	22
1.5 The Help Command . . . . .	24
1.6 Executing GemStone Smalltalk Expressions . . . . .	24
1.7 Escaping to an Editor . . . . .	25
1.8 Controlling the Display of Results . . . . .	26
Display Level . . . . .	26
Setting Limits on Object Displays . . . . .	27
Displaying Variable Names, OOPs, and Hex Byte Values . . . . .	28
Instance Variable Names . . . . .	28
Hexadecimal Byte Values. . . . .	29
OOP Values . . . . .	29
1.9 Creating and Changing Methods . . . . .	30
Editing Methods. . . . .	32

1.10 Listing Methods and Categories. . . . .	32
1.11 Committing and Aborting Transactions . . . . .	33
1.12 Capturing Your Topaz Session In a File . . . . .	33
1.13 Filing Out Classes and Methods. . . . .	35
1.14 Creating a Topaz Script for Batch Processing . . . . .	36
1.15 Taking Topaz Input from a File . . . . .	37
1.16 Interrupting Topaz and GemStone . . . . .	38
1.17 Multiple Concurrent GemStone Sessions. . . . .	38
1.18 Structural Access To Objects . . . . .	40
Examining Instance Variables with Structural Access . . . . .	41
Specifying Objects . . . . .	42
Object Identity Specification Formats . . . . .	42
Literal Object Specification Formats. . . . .	42
Specifying Method Selectors. . . . .	43
1.19 Defining Local Variables . . . . .	44
Creating Variables . . . . .	44
Displaying Current Variable Definitions. . . . .	45
Clearing Variable Definitions . . . . .	46
1.20 Sending Messages. . . . .	46
1.21 Logging Out . . . . .	47
1.22 Leaving Topaz. . . . .	47

## ***Chapter 2. Debugging Your GemStone Smalltalk Code*** **49**

2.1 Step Points and Breakpoints. . . . .	50
2.2 Setting, Clearing, and Examining Breakpoints . . . . .	51
2.3 Examining the GemStone Smalltalk Call Stack . . . . .	53
Proceeding After a Breakpoint . . . . .	55
Examining and Modifying Temporaries and Arguments . . . . .	55
Select a Context for Examination and Debugging. . . . .	57

## ***Chapter 3. Command Dictionary*** **59**

ABORT. . . . .	60
BEGIN . . . . .	61
BREAK aSubCommand . . . . .	62

Method Breakpoints . . . . .	62
Disabling and Enabling Breakpoints . . . . .	63
CATEGORY: aCategoryName. . . . .	65
CLASSMETHOD[: aClassName] . . . . .	66
COMMIT. . . . .	67
CONTINUE [anObjectSpec] . . . . .	68
DEFINE [aVarName [anObjectSpec [aSelectorOrArg]...]] . . . . .	69
DISASSEM [aClassParameter] aParamValue . . . . .	70
DISPLAY aDisplayFeature. . . . .	71
DOIT . . . . .	73
DOWN [anInteger] . . . . .	74
DOWNR [anInteger] . . . . .	75
DOWNRB [anInteger] . . . . .	75
EDIT aSubCommandOrSelector [aSelector] . . . . .	76
Creating or Modifying Blocks of GemStone Smalltalk Code . . . . .	76
Creating or Modifying GemStone Smalltalk Methods . . . . .	76
ERRORCOUNT . . . . .	78
EXIT [aSmallInt   anObjectSpec] . . . . .	79
EXITIFNOERROR. . . . .	80
EXPECTBUG bugNumber . . . . .	81
EXPECTERROR anErrorCategory anErrorNumCls. . . . .	82
EXPECTVALUE anObjectSpec [anInt] . . . . .	85
FILEOUT aSubCommandOrSelector [TOFILE: aFileName] . . . . .	87
FR_CLS [anInteger] . . . . .	88
FRAME [anInteger] . . . . .	89
GCITRACE aFileName. . . . .	90
HELP [aTopicName] . . . . .	91
HIERARCHY [aClassName]. . . . .	92
HISTORY [anInteger]. . . . .	93
IFERR bufferNumber [aTopazCommandLine] . . . . .	94
IFERR_CLEAR . . . . .	95
IFERR_LIST . . . . .	96
IFERROR [aTopazCommandLine] . . . . .	97
IMPLEMENTORS selectorSpec . . . . .	98
INPUT [aFileName   POP] . . . . .	99
INSPECT [anObjectSpec]. . . . .	100
INTERP . . . . .	101
INTERPENV. . . . .	102
LEVEL anIntegerLevel . . . . .	103
LIMIT [BYTES   OOPS] anInteger . . . . .	104
LIST. . . . .	105

Browsing Dictionaries and Classes . . . . .	105
Listing Methods . . . . .	106
Listing Step Points . . . . .	106
Listing Breakpoints . . . . .	107
LISTW . . . . .	109
LOADUA aFileName . . . . .	110
LOGIN . . . . .	111
LOGOUT . . . . .	113
LOGOUTIFLOGGEDIN . . . . .	114
LOOKUP (METH   METHOD   CMETH   CMETHOD) selectorSpec . . . . .	115
LOOKUP className [CLASS] selectorSpec . . . . .	115
Finding and Listing Methods . . . . .	115
METHOD[: aClassName] . . . . .	117
NBRESULT . . . . .	118
NBRUN . . . . .	119
NBSTEP . . . . .	120
OBJ1 anObjectSpec . . . . .	121
OBJ2 anObjectSpec . . . . .	121
OBJ1Z anObjectSpec . . . . .	122
OBJ2Z anObjectSpec . . . . .	122
OBJECT anObjectSpec [AT: anIndex [PUT: anObjectSpec]] . . . . .	123
OMIT aDisplayFeature . . . . .	125
OUTPUT (PUSH   APPEND   PUSHNEW   POP) aFileName [ONLY] . . . . .	127
PAUSEFORDEBUG [errorNumber] . . . . .	129
PKGLOOKUP (METH   METHOD   CMETH   CMETHOD) selectorSpec . . . . .	130
PKGLOOKUP className [CLASS] selectorSpec . . . . .	130
PRINTIT . . . . .	131
PROTECTMETHODS . . . . .	132
QUIT [aSmallInt   anObjectSpec]. . . . .	133
RELEASEALL . . . . .	134
REMARK commentText . . . . .	135
REMOVEALLCLASSMETHODS [aClassName] . . . . .	136
REMOVEALLMETHODS [aClassName] . . . . .	137
RUBYCLASSMETHOD . . . . .	138
RUBYHIERARCHY . . . . .	138
RUBYIMPLEMENTORS . . . . .	138
RUBYLIST . . . . .	138
RUBYLOOKUP . . . . .	138
RUBYMETHOD . . . . .	138
RUBYRUN . . . . .	138
RUN . . . . .	139

RUNENV . . . . .	140
SEND anObjectSpec aMessage . . . . .	141
SENDENV . . . . .	142
SENDERS selectorSpec . . . . .	143
SET aTopazParameter [aParamValue] . . . . .	144
SHELL [aHostCommand] . . . . .	149
SPAWN [aHostCommand] . . . . .	150
STACK [aSubCommand] . . . . .	151
Display the Active Call Stack . . . . .	151
Display or Redefine the Active Context . . . . .	153
Save or Delete the Active Call Stack During Execution . . . . .	154
Display All Call Stacks . . . . .	154
Redefine the Active Call Stack . . . . .	154
Remove Call Stacks . . . . .	155
STATUS . . . . .	156
STEP (OVER   INTO) . . . . .	157
STK [aSubCommand] . . . . .	158
STRINGS selectorSpec . . . . .	159
STRINGSIC selectorSpec . . . . .	160
SUBCLASSES [aClassName] . . . . .	161
TEMPORARY [aTempName[/anInt] [anObjectSpec] ] . . . . .	162
THREAD [anInt] [CLEAR]. . . . .	164
THREADS [CLEAR] . . . . .	165
TIME . . . . .	166
TOPAZPAUSEFORDEBUG [errorNumber]. . . . .	167
TOPAZWAITFORDEBUG . . . . .	167
UNPROTECTMETHODS . . . . .	168
UP [anInteger]. . . . .	169
UPR [anInteger]. . . . .	170
UPRB [anInteger]. . . . .	170
WHERE [anInteger   aString]. . . . .	171
WHRB [anInteger] . . . . .	172
WHRUBY [anInteger] . . . . .	172

## ***Appendix A. Topaz Command-Line Syntax*** **173**

A.1 Command-Line Syntax . . . . .	173
A.2 Options . . . . .	174

<i>Appendix B. Network Resource String Syntax</i>	<b>177</b>
B.1 Overview . . . . .	177
B.2 Defaults . . . . .	178
B.3 Notation . . . . .	179
B.4 Syntax . . . . .	180
 <i>Index</i>	 <b>183</b>

# *Getting Started with Topaz*

---

Topaz is a GemStone programming environment that provides keyboard command access to the GemStone system. Topaz does not require a windowing system and so is the interface of choice for batch work and for many system administration functions.

This chapter explains how to run Topaz and how to use some of the most important Topaz commands. Chapter 3 provides descriptions of all Topaz commands.

To run Topaz, your system administrator or GemStone data curator must first have installed GemStone/S 64 Bit version 3.0 on your system. You must have an running repository monitor (Stone), and, to run the remote version of Topaz, an accessible network service process (NetLDI). The *GemStone/S 64 Bit Installation Guide* explains how to install these components.

Your environment must contain a definition of the `$GEMSTONE` variable and your execution path must include the GemStone binary directory (`$GEMSTONE/bin` on UNIX systems, `%GEMSTONE%\bin` on Windows). Consult your system administrator or GemStone data curator if you need help with this.

Examples throughout this book were created on a UNIX system. Behavior and appearance of Topaz on Windows systems is the same, except where noted.

## GemStone Smalltalk and Ruby

Topaz is designed to support the GemStone Ruby environment (MAGLEV) as well as GemStone/S 64 Bit version 3.0. A number of Topaz commands and options are provided for use with Ruby but are not used in GemStone Smalltalk installations. These commands are identified as such in Chapter 3, Command Dictionary.

The `environmentId` specifies a method lookup environment that is primarily used in Ruby environments, although it may be available for Smalltalk applications. By default, it is 0 in Smalltalk applications. The `environmentId` will be visible in method displays when debugging.

## 1.1 Invoking Topaz

To invoke Topaz, simply type **topaz** on the command line. The program responds by printing its copyright banner and issuing a prompt, as shown in Figure 1.1.

**Figure 1.1 Topaz Banner and Prompt**

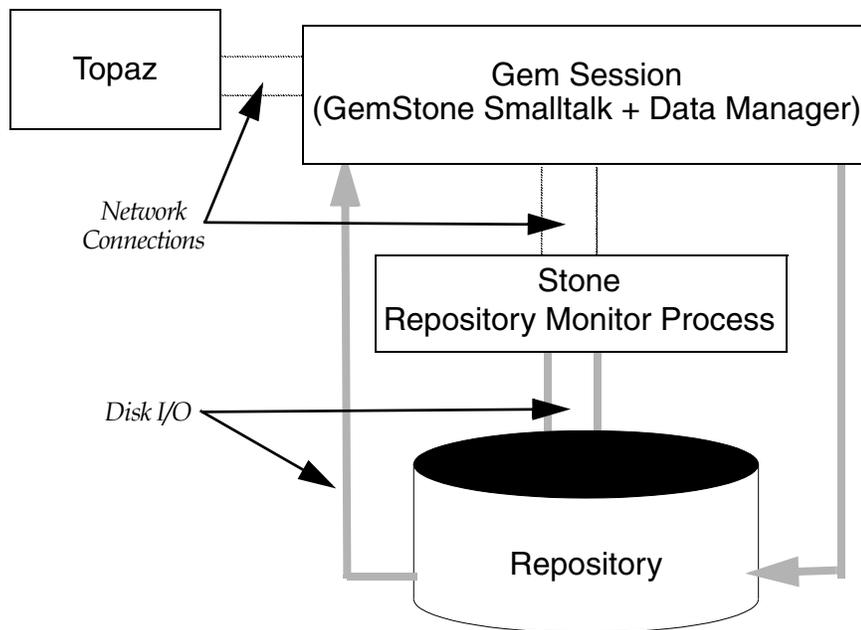
```
% topaz
|
|      GemStone/S64 Object-Oriented Data Management System
|      Copyright (C) VMware, Inc. 1986-2011.
|      All rights reserved.
|      Covered by U.S Patents:
|      6,256,637 Transactional virtual machine architecture
|      6,360,219 Object queues with concurrent updating
|      6,567,905 Generational Garbage Collector.
| 6,681,226 Selective Pessimistic Locking for a Concurrently Updateable Database
+-----+
| PROGRAM: topaz, Linear GemStone Interface (Remote Session)
| VERSION: 3.0.0, Tue Jun 14 12:57:05 US/Pacific 2011
| BUILD: 64bit-25988
| BUILT FOR: SPARC (Solaris)
| MODE: 64 bit
| RUNNING ON: 2-CPU handel sun4u (Solaris 2.10 Generic_142900-04) 2750MHz
| sparcv9,2048MB
| PROCESS ID: 27630      DATE: 06/17/11 15:41:54 PDT
| USER IDS: REAL=gsuser (531) EFFECTIVE=gsuser (531)
|
neither .topazini nor $HOME/.topazini were found
topaz>
```

## 1.2 Overview of a GemStone Session

A GemStone session consists of four parts, as shown in Figure 1.2. These are:

- An **application**, in this case, Topaz.
- One **repository**. An application has one repository to hold its persistent objects.
- One repository monitor, or **Stone** process, to control access to the repository.
- At least one GemStone session, or **Gem** process. All applications, including Topaz, must communicate with the repository through Gem processes. A Gem provides a work area within which objects can be used and modified. Several Gem processes can coexist, communicating with the repository through a single Stone process.

Figure 1.2 GemStone Object Server Components



## 1.3 Remote Versus Linked Versions

In Figure 1.1, notice that the Topaz startup banner's `PROGRAM` line refers to `Remote Session`. Two versions of Topaz are available to you: *remote procedure call* (or *RPC*) and *linked*. Unless you specify otherwise, the `topaz` command invokes the *RPC* version. The *RPC* version of Topaz allows you to run multiple *RPC* Topaz sessions. These run separately from their *Gem* processes, so you can run Topaz and the *Gem* processes on separate nodes.

The `topaz -l` (for *linked*) command line invokes the *linked* version of Topaz. The *linked* version allows you to run multiple Topaz sessions, but session number one is always a *linked* session, where the Topaz session and a *Gem* exist as a single process. Any additional sessions are *RPC*. *Linked* sessions do not require a *NetLDI* to be running.

Under Windows, only the *RPC* version of Topaz is available, and the *Gem* sessions must be run on a server platform.

The examples in this chapter can be executed equally well from either *linked* or *RPC* Topaz. For additional command-line options, see Appendix A.

## 1.4 Logging In to GemStone

The first step in establishing a connection to GemStone and logging in is to give Topaz some information about the GemStone repository you will be using. To log in to the repository you must provide a GemStone user name and password. If you are running the *RPC* version of Topaz, you also need to provide your operating system user name and password for the host on which your GemStone session resides.

Here are the parameters to be established to log in to GemStone through Topaz:

- **The GemStone name** (`gemstone`). This the name of the Stone process to use and, optionally, the name of the network node on which it resides. The default name is `gs64stone`. If your Stone process is named `gs64stone` and is running on the local node, and the *Gem* process will also run on the local node, you don't have to set the GemStone name.

Otherwise, specify the name of the Stone. If the node where the Stone is running is not the one where the *Gem* will run, you also need the name of the Stone host and perhaps the type of network connection between the Stone and *Gem* hosts. To specify a process named `gs64stone` running on node `central`, you can use a network resource string of the form

!@central!gs64stone. Your GemStone data curator can give you the exact string to use. Appendix B describes the syntax of network resource strings.

- **Your GemStone user name and password** (username). Your GemStone data curator can give you these.
- **Your host user name and password** (hostusername and hostpassword). The name and password that you use when you log in to the host operating system. These are needed only for RPC sessions.
- **The GemStone service name** (gemnetid). For the RPC version the default is gemnetobject. You may also use gemnetdebug, if you are debugging memory issues, or create a custom gemnetobject service.

For the linked version of Topaz, do not set the gemnetid, or set the gemnetid to ". If you set it to gemnetobject, all your sessions will be RPC, in spite of having invoked the linked version of topaz. You can use a network resource string of the form !@central!gemnetobject to start a Gem process on a remote node.

Your GemStone data curator can assist you if you encounter difficulties.

In general, you can abbreviate any Topaz command to uniqueness. Topaz commands (such as **set gemnetid** and **login**) are case-insensitive. The arguments you specify, however, must meet your operating system's requirements for capitalization and spelling.

Use the Topaz **set** command to establish these parameters. For example:

```
topaz> set gemstone gs64stone
topaz> set username 'Isaac Newton'
```

Whenever a Topaz parameter such as "Isaac Newton" contains white space, it must be enclosed within single quotes.

This is sufficient for the linked logins. If you are logging in RPC, you must also provide the following information:

```
topaz> set gemnetid gemnetobject
topaz> set hostusername 'newtoni'
topaz> set hostpassword
Host Password? (Type your host password; it won't be echoed)
```

To see your current login settings and other information about your Topaz session, type **status**:

```
topaz> status
```

```
Current settings are:
display level: 0
omit oops
omit bytes
display instance variable names
omit automatic result checks
omit interactive pause on errors
listwindow: 20
stackpad: 45
tab (ctl-H) equals 8 spaces when listing method source
using line editor
  line editor history: 100
  topaz input is from a tty on stdin
EditorName_____ vi

CompilationEnv____ 0
Connection Information:
UserName_____ 'Isaac Newton'
Password_____ (not set)
HostUserName____ 'newtoni'
HostPassword____ (set)
GemStone_____ 'gs64stone'
GemStone NRS____
'!#encrypted:newtoni@password#server!gs64stone'
GemNetId_____ 'gemnetobject'
GemNetId NRS____
'!#encrypted:newtoni@password!gemnetobject'
CacheName_____ (default)
```

If you are using the linked version of Topaz, certain login parameters (HostUserName and HostPassword) have no effect. Setting GemNetId will result in an RPC login.

If any login settings are incorrect, use the **set** command to fix them.

You are now ready to issue the **login** command, connecting your Topaz session to the GemStone repository:

```
topaz> login
GemStone password? (type your GemStone password)
[05/23/11 17:50:36.17 PDT]
gci login: currSession 1 rpc gem processId 27623 OOB keep-
alive interval 0
successful login
topaz 1>
```

As this example shows, Topaz displays a session number in its prompt once you have logged in.

You are also free to supply several of these login parameters on a single command line in any order, and to abbreviate the parameter names:

```
topaz> set gemstone gs64stone user 'Isaac Newton'
topaz> set gemnetid gemnetobject hostuser 'newtoni'
topaz> set hostpass <return>
Host Password? (type your host password)
topaz> login
[05/23/11 17:46:19.591 PDT]
gci login: currSession 1 rpc gem processId 27612 OOB keep-
alive interval 0
successful login
topaz 1>
```

Because setting the host user name causes Topaz to discard the current host password, you must set **hostusername** before **hostpassword**.

If you are using the linked version of Topaz, you can login with fewer **set** commands:

```
topaz> set gemstone gs64stone user 'Isaac Newton' pass gravity
topaz> login
[Info]: LNK client/gem GCI levels = 842/842
[Info]: User ID: 'Isaac Newton'
[Info]: Repository: gs64stone
[Info]: Session ID: 5
[Info]: GCI Client Host: <Linked>
[Info]: Page server PID: -1
[Info]: Login Time: 05/21/11 16:03:17.187 PDT
[05/21/11 16:03:17.187 PDT]
  gci login: currSession 1 rpc gem processId -1 OOB keep-
alive interval -1
successful login
topaz 1>
```

## Setting Up a Login Initialization File

You can streamline the login process by creating an initialization file that contains the **set** commands needed for logging in. When you invoke Topaz, it automatically executes those commands for you. If you insert **set hostpassword** and **login** commands without parameters, Topaz automatically prompts you for the necessary values.

**Table 1.1 Topaz Initialization File Names**

Platform	Name of Topaz Initialization File	Expected Location
UNIX	.topazini	Current directory, then user's home directory
Windows	topazini.tpz	Current directory, then user's home directory. If home directory is undefined, uses home directory of the account that started Windows, if any, or DRIVE:\users\default where DRIVE is the local device on which Windows is installed.

You may also explicitly specify a path for a topazini file on the command line where you started up the Topaz executable. Using this option overrides any topazini files that Topaz would otherwise use.

```
% topaz -I /gemstone/utils/mylogin.topazini
```

If you want to run Topaz non-interactively, you must explicitly specify both the GemStone and host passwords in this initialization file.

**CAUTION:**

*Entering your passwords in a file can pose a security risk.*

The Topaz initialization file shown in Figure 1.3 performs most of the same functions as the interactive commands shown in the previous discussion.

**Figure 1.3 Topaz Initialization File**

```
set gemstone gs64stone
set gemnetid gemnetobject
set username 'Isaac Newton'
set password mypassword
set hostusername 'newtoni'
set hostpassword hostpassword
login
```

If you have an initialization file, to start Topaz without using the initialization file, use the **-i** option. See Appendix A.

Once Topaz has read an initialization file like this one, logging in is quite simple; you are automatically logged on when you start the topaz application.

You can also store host user account information in a network initialization file, `.netrc`. If you do not explicitly supply a host username and password, on UNIX hosts Topaz tries to find a username and password for the designated node in the file `$HOME/.netrc`. If that file is properly configured, you won't need to explicitly supply a host user name and password. For information about how to configure your network initialization file, see the discussion of **set hostusername** in Chapter 3 of this manual.

If you choose not to include your password in an initialization file, Topaz will start up with the following prompt.

```
GemStone Password?      Type your password. It will not be echoed.
topaz 1>
```

## 1.5 The Help Command

You can type **help** at the Topaz prompt for information about any Topaz command. For example:

```
topaz 1> help exit
```

```
EXIT [<status>]
```

Terminates Topaz, returning to the parent process or operating system. If you are still logged in to GemStone when you type EXIT, this will abort your transaction and log out all active sessions. Although you can abbreviate most other Topaz commands and parameter names, EXIT must be typed in full.

If this command has a <status> argument and it is an integer, then the integer is used as the exitStatus. If the status argument is an object specification that resolves to an integer, then that value is used as the exitStatus. Only 8 bits of the integer are returned, exitStatus should be 0..255. If it is not an integer then an error is returned and the process does not exit.

If the command does not have a <status> argument, the exitStatus will be either 1 if there were GCI errors or the topaz errorcount is not zero, or 0 if no errors had occurred.

EXIT has the same function as QUIT.

```
Topic?          (press Return to exit the help utility)
topaz 1>
```

## 1.6 Executing GemStone Smalltalk Expressions

By following the examples in the rest of this chapter, you'll learn how to create and execute GemStone Smalltalk code, and how to inspect GemStone objects. If you need to log out of your session before you finish, you can use the **commit** command to save the classes and methods you have created. To start where you

left off in a new session, you will have to reset the current class and category, but usually not the default screen display settings.

Once you've logged in to GemStone, you can execute Smalltalk expressions with the **printit** command. The following use of **printit**, for example, creates an instance of `DateTime` representing the current Date and Time:

```
topaz 1> printit
DateTime now
%
a DateTime
  year          2011
  dayOfYear     108
  milliseconds  82384763
  timeZone      a TimeZone
```

All of the lines after the **printit** command and before the first line in which `%` is the first character are sent to GemStone for execution as GemStone Smalltalk code. Topaz then displays the result and prompts you for a new command.

If there is an error in your code, Topaz displays an error message instead of a legitimate result. You can then retype the expression with errors corrected, or use the Topaz **edit** function to correct and refine the expression.

## 1.7 Escaping to an Editor

To use the **edit** function, you must first have established the name of the host editor you wish to use. Topaz can read the UNIX environment variable `EDITOR`, if you have it set. Otherwise, use the Topaz **set editorname** command, interactively or in your Topaz initialization file.

```
topaz 1> set editorname vi
```

Then, to edit the text of the last **printit** command, you need only do this:

```
topaz 1> edit last
```

Topaz opens your editor, as a subprocess, on the text of the last **printit** command. When you exit the editor, Topaz saves the edited text in a temporary file and asks you whether you'd like to compile and execute the altered code. If you type **yes**, Topaz effectively reissues your **printit** command with the new text.

To use the editor for creating an entirely new block of code for execution, use **edit new text** instead of **edit last**.

See "Editing Methods" on page 32 for more on **edit**.

## 1.8 Controlling the Display of Results

Topaz provides several commands that let you control the amount and kind of information it displays about results.

### Display Level

When Topaz displays a result object, it always displays the object itself, but the display of the name and value of each instance variable is controlled by the level, and the particular command used to execute the code.

```
topaz 1> printit
DateTime now
%
a DateTime
  year          2011
  dayOfYear     108
  milliseconds  82727699
  timeZone      a TimeZone
```

This display is one level deep: the instance variables are displayed, but not the instance variables of any complex objects in the instance variable values.

The **printit** command always displays results with one level, as shown above. The **doit** command displays 0 levels:

```
topaz 1> doit
DateTime now
%
18/04/2011 16:01:29
```

You can use the **level** and **run** commands to ask for more or less information about results. The **run** command executes code and displays the results according to the most recent **level** execution. For example, with level 0, the **run** command produces the same display as the **doit** command:

```
topaz 1> level 0
topaz 1> run
DateTime now
%
18/04/2011 16:01:29
```

Setting the level to 2 would give this view:

```
topaz 1> level 2
topaz 1> run
DateTime now
%
a DateTime
  year                2011
  dayOfYear           108
  milliseconds        83026019
  timeZone             a TimeZone
    transitions        an Array
    leapSeconds        nil
    stream             nil
    types              nil
    charcnt            nil
    standardPrintString PST
    dstPrintString     PDT
    dstStartTimeList  an IntegerKeyValueDictionary
    dstEndTimeList    an IntegerKeyValueDictionary
```

As you can see, setting the display level to 2 causes Topaz to display each instance variable within each of class `Animal`'s instance variables. The maximum display level is 32767.

If the display level setting is high enough and the object to be displayed is cyclic (that is, if it contains itself in an instance variable), Topaz will faithfully follow the circularity, displaying the object repeatedly.

## Setting Limits on Object Displays

The **limit bytes** command controls how much Topaz displays of a byte object (instance of `String` or one of `String`'s subclasses) that comes back as a result. Similarly, **limit oops** controls how much Topaz displays of pointer or NSC objects that come back as a result.

By default, Topaz attempts to display all of a result, no matter how long. The following example shows how you could use **limit bytes** to make Topaz limit the display to the first 4 bytes:

```
topaz 1> limit bytes 4
topaz 1> printit
  'this and that'
%
this
```

Setting the limit to 0 restores the default condition.

## Displaying Variable Names, OOPs, and Hex Byte Values

Two complementary commands, **display** and **omit**, control the display of instance variable names, hexadecimal byte values, and OOPs (the *object-oriented pointers* that uniquely identify GemStone objects internally).

### Instance Variable Names

As you saw in the display of class `Animal`, Topaz normally prints the name of each named instance variable with its value. If you don't need this information, you can speed up the display of results by telling Topaz to **omit names**, as in the following example:

```
topaz 1> omit names
topaz 1> printit
DateTime now
%
a DateTime
  i1 2011
  i2 108
  i3 83686097
  i4 a TimeZone
```

Entering **display names** restores Topaz to the default condition.

## Hexadecimal Byte Values

Topaz ordinarily displays byte objects such as Strings literally, with no additional information. If you enter **display bytes** Topaz includes the hexadecimal value of each byte. For example:

```
topaz 1> display bytes
topaz 1> printit
  'this and that'
%
1 'this and that' 74 68 69 73 20 61 6e 64 20 74 68 61 74
```

Entering **omit bytes** restores the default byte display mode.

## OOP Values

It's occasionally useful in debugging to examine the numeric object identifiers that GemStone uses internally. If you tell Topaz to **display oops**, it prints a bracketed object header with each object, which looks like this:

```
[21336065 sz:3 cls: 110849 Symbol]
```

Each object header contains:

- The object's OOP (a 64-bit unsigned integer)
- the object's size, calculated by summing all of its named, indexed, and unordered instance variable fields
- the OOP of the object's class and the class name

For example:

```
topaz 1> display oops
topaz 1> printit
DateTime now
%
[21716225 sz:4 cls: 118785 DateTime] a DateTime
  year      [16090 sz:0 cls: 74241 SmallInteger] 2011 == 0x7db
  dayOfYear [866 sz:0 cls: 74241 SmallInteger] 108 == 0x6c
  milliseconds [669886402 sz:0 cls: 74241 SmallInteger] 83735800 == 0x4fdb4f8
  timeZone   [8668161 sz:9 cls: 13228033 TimeZone] a TimeZone
```

You can turn off the display of OOPs by typing **omit oops** at the Topaz prompt.

## 1.9 Creating and Changing Methods

Creating a class is done using GemStone Smalltalk class creation protocol. For more on class creation, refer to the *GemStone Programming Guide*.

For example,

```
topaz 1> printit
Object subclass: 'Animal'
  instVarNames: #('name' 'favoriteFood' 'habitat')
  classVars: #()
  classInstVars: #()
  poolDictionaries: #()
  inDictionary: UserGlobals
%
a metaAnimal
  superClass          a metaObject
  format              0
  instVarsInfo        1125899906846723
  instVarNames        an Array
  constraints          an Array
  classVars           nil
  methDicts           a GsMethodDictionary
  poolDictionaries    nil
  categorys           a GsMethodDictionary
  primaryCopy         nil
  name                Animal
  classHistory         a ClassHistory
  transientMethDicts  an Array
  destClass           nil
  timeStamp           a DateTime
  userId              DataCurator
  extraDict           a SymbolDictionary
  classCategory       nil
  subclasses          nil
```

Once you have a class, you can create methods for it in topaz. The first step in creating or editing a method is to tell Topaz the name of the method's class. Do this with the **set class** command:

```
topaz 1> set class Animal
```

This establishes a context for your subsequent work so that you don't need to supply the class name each time you create or edit a method.

Similarly, you'll need to supply the name of the method category in which you want to work:

```
topaz 1> set category Updating
```

If the category you name doesn't exist, Topaz creates it when you first compile a method.

Topaz maintains this information about the current class and category until you explicitly change it. You can examine your current class and category settings by typing **status**.

```
topaz 1> status
```

Current settings are:

*(display of current settings and connection information appears here)*

browsing information:

```
Class_____ Animal
Category_____ Updating
Source String Class__ String
```

Once you've established a class and a category, you can begin an instance method definition by issuing the **method:** command at the Topaz prompt:

```
topaz 1> method: ^
habitat: newValue
"Modify the value of the instance variable 'habitat'."
  habitat := newValue
%
```

The **method:** command takes a single argument: the name of the class for which the method will be compiled. As shown here, wherever Topaz expects the name of a class, you can simply type a caret (^) to tell Topaz to use the current class (in this case, Animal).

A class method definition is similarly initiated by the Topaz command **classmethod:**. For example:

```
topaz 1> classmethod: ^
returnAString
"Returns an empty String"
^String new
%
```

Like the text of a **printit** command, the text of a method definition is terminated by the first line that starts with the % character.

As soon as you enter the %, Topaz sends the method's text to GemStone for compilation and inclusion in the selected class and category.

## Editing Methods

You can debug and refine methods by using Topaz's **edit** function in much the same way you use that function to create and modify **printit** commands. For example, to edit the existing instance method `habitat:` in the current class, you would enter **edit** as shown below:

```
topaz 1> edit method habitat:
```

Here is how you would edit an existing class method:

```
topaz 1> edit classmethod returnAString
```

To create an entirely new method with the editor, you can enter **edit new method** or **edit new classmethod**.

If you omit the **method** and **classmethod** keywords, you must specify an instance method to be edited; for example, **edit habitat:**.

## 1.10 Listing Methods and Categories

If you need to see which categories and methods are in the current class, use the Topaz **list** command. The command **list categoriesIn:** causes Topaz to list all of the class and instance method selectors in the selected class by category.

To list the source code of an instance method, type **list method:** *aMethodName* as in the following example:

```
topaz 1> list method: habitat:  
habitat: newValue  
"Modify the value of the instance variable 'habitat'."  
habitat := newValue
```

A parallel command, **list classmethod:**, lists the source of the given class method. If you omit the keywords **method:** and **classmethod:** from your **list** command, you must specify an instance method you wish to list.

Other **list** options allow you to examine the classes in one or all of your symbol list dictionaries or to examine the methods in some class other than the current class. For more information, see the description of **list** in Chapter 3 of this manual.

## 1.11 Committing and Aborting Transactions

In GemStone, each session's operations normally exist in a transaction that maintains a temporary, private workspace. The **commit** command ends your current transaction and stores this information in the repository, for use in later sessions and by other users.

To commit a transaction while using Topaz, you can execute the GemStone Smalltalk expression `System commitTransaction` within a **printit** command, or you can enter the Topaz **commit** command:

```
topaz 1> commit  
Successful commit
```

Similarly, you abort a transaction by executing the GemStone Smalltalk expression `System abortTransaction` within a **printit** command, or by entering **abort** at the Topaz command prompt. Entering **abort** does not reset Topaz system definitions, such as your current class and category.

Although you can abbreviate most other Topaz commands and parameter names, **commit**, **abort**, **logout**, and **exit** (the last two of which implicitly abort your transaction) must be typed in full.

## 1.12 Capturing Your Topaz Session In a File

It's often useful to keep a record of your interactions with GemStone during testing and debugging. You might also want to record a typical series of GemStone operations that could be used as a training guide or edited into a batch processing file.

You can do this with the Topaz command **output push**. This command causes Topaz to write all input and output to a named file as well as to standard input and standard output (your terminal).

The following example causes all subsequent interactions to be captured in a file called `animaltest.log`:

```
topaz 1> output push animaltest.log
```

If the file you name doesn't exist, Topaz creates it. Under UNIX, if you name an existing file, Topaz overwrites it.

To add output to an existing file without losing its current contents, precede the file name with an ampersand (&). For example:

```
topaz 1> output push &animaltest.log
```

The following example stops output to the current file:

```
topaz 1> output pop
```

As the command names **push** and **pop** imply, Topaz can maintain a stack of up to 20 output files. If you add the keyword **only** to the **push** command lines, current interactions are captured only in the file on top of the stack. This prevents the results from showing on your screen, however.

```
topaz 1> output push animaltest2.log only
```

Otherwise, the output is duplicated in each file on the stack. For example, the following sequence would capture one command in the file `mathtest.log`, and a second command in `mathtest2.log`:

```
topaz 1> remark Capture the next command
topaz 1> remark and result in mathtest.log
topaz 1> output push mathtest.log
topaz 1> printit
5 * 8
%
40
topaz 1> remark Capture the next command
topaz 1> remark and result in mathtest2.log
topaz 1> output push mathtest2.log only
topaz 1> printit
5 * 9
%
topaz 1> remark Close mathtest2.log
topaz 1> remark and resume using mathtest.log
topaz 1> output pop
```

Notice that the result of the second command, 45, did not appear on the screen. If the second **push** command line did not have the **only** keyword, the entire sequence would have been recorded in `mathtest.log`, and the second command duplicated in `mathtest2.log`.

Also notice the use of **remark** in this example—you can use either **remark** or an exclamation point in column 1 to begin a comment. Comments are often useful for annotating Topaz input files created for batch processing or testing.

## 1.13 Filing Out Classes and Methods

Sometimes you'll want to create, edit, or archive a class and some large fraction of its methods as a monolithic chunk of source code. This makes it possible to:

- transport your code to other GemStone systems,
- perform global edits and recompilations,
- produce paper copies of your work, and
- recover code that would otherwise be lost when you are unable to commit.

The Topaz **fileout** command can create an executable Topaz script defining a class and/or any or all of the class's methods. You can process the script using editors or other operating system utilities and then execute it with the Topaz **input** command. The following command:

```
topaz 1> fileout class: Animal toFile: animal.gs
```

would create in the file `animal.gs`, a Topaz script containing a definition of class `Animal` and all of its categories and methods. Here is how `animal.gs` would look:

```
printit
Object subclass: 'Animal'
  instVarNames: #( 'name' 'favoriteFood' 'habitat')
  classVars: #()
  classInstVars: #()
  poolDictionaries: {}
  inDictionary: UserGlobals
%
category: 'Updating'
method: Animal
habitat: newValue

"Modify the value of the instance variable 'habitat'."
habitat := newValue
%
...
```

"Filing in" this script with the **input** command would create a new class `Animal` exactly like the original.

In addition to **class**, the **fileout** command has four other subcommands:

**fileout category:**

Files out all the methods in the named category.

**fileout classcategory:**

Files out all the class methods in the named category.

**fileout classmethod:**

Files out the source code of the method identified in the argument by its selector.

**fileout method:**

Files out the specified instance method. You can omit the **method:** portion of a **fileout** command, unless the instance method's selector is also the name of one of the other **fileout** subcommands. For example, to file out a method named **habitat**, you could simply enter

```
topaz 1> fileout habitat
```

To file out a method named **category:**, however, you would need to enter

```
topaz 1> fileout method: category:
```

## 1.14 Creating a Topaz Script for Batch Processing

Just as the **fileout** command creates an executable Topaz script defining a class, you can create your own Topaz script that performs any series of GemStone operations. If you have complicated queries or a long series of repository updates that you repeat on a regular basis, this is an easy way to do it. You can type the Topaz commands into a file, test and edit them until they run with no errors, and then you have a script that will do automatic batch processing for you. If your procedure changes slightly from day to day, you can easily edit the script. Because the files duplicate what you would do interactively, they are also useful training tools.

Another way to produce such a script is to capture a typical Topaz session in a file, using **output push**. Edit the output file to remove the prompts and results, leaving only the Topaz commands and GemStone Smalltalk code. For example, suppose

you wanted to make a script of the `mathtest2.log` file you created earlier. This is how it looks:

```
topaz 1> printit
5 * 9
%
45
topaz 1> remark Close mathtest2.log
topaz 1> remark and resume using mathtest.log
topaz 1> output pop
```

To make it an executable script, remove the prompts, results, and unnecessary commands, and make the comments helpful.

```
remark This multiplies two numbers
printit
5 * 9
%
```

Do not use the `edit` command for batch processing. Instead, use the `method:` and `classmethod:` commands to create methods in batch processes, and the `printit` or `doit` commands to execute blocks of code in batch.

## 1.15 Taking Topaz Input from a File

Although Topaz ordinarily takes its input from standard input (usually your terminal), you can use the `input` command to make Topaz take its input from a file. The following command, for instance, would make Topaz read and execute the commands in a file called `animal.gs` in your UNIX `$HOME` directory:

```
topaz 1> input $HOME/animal.gs
```

The UNIX environment variable `$HOME` is expanded to the full filename before the `input` command is carried out.

Batch processing goes very quickly. It is a good idea to use `output push` to record the session, so you can check for errors.

## 1.16 Interrupting Topaz and GemStone

Three kinds of interruption (break) using **Ctrl-C** are possible when you're using Topaz:

- When Topaz is awaiting input from your terminal, such as when you're entering a command, you can enter **Ctrl-C** to terminate entry of the command and prepare Topaz for accepting a new command.
- When GemStone is compiling or executing some GemStone Smalltalk code sent to it by Topaz, such as in a **printit** command, typing **Ctrl-C** sends a request to GemStone to interrupt its activities as soon as possible. GemStone stops execution at the conclusion of the current method, and Topaz displays the message: `A soft break was received.`
- Typing **Ctrl-C** three times immediately halts Topaz. Do this only in an emergency. All GemStone work performed since you last committed is lost.

## 1.17 Multiple Concurrent GemStone Sessions

Topaz can keep several independent GemStone sessions alive simultaneously. This allows you to switch from one session to another, for instance to access more than one GemStone repository. Both RPC and linked versions of Topaz allow you to run multiple sessions by using the **login** and **set session** commands; however, you can only have one linked session at a time.

The following example shows how you might create a second session, make the new session your current session, then return to the original session.

```
topaz> login
gci login: currSession 1 rpc gem processId 95
successful login
topaz 1> set gemstone !@srv2!gs64stone
topaz 1> set username isaac
Warning: GemStone is clearing previous GemStone password.
GemStone password? <password typed here but not echoed>
topaz 1> login
gci login: currSession 2 rpc gem processId 141
successful login
topaz 2> printit
UserGlobals at: #myVar put: 1
%
1
topaz 2> set session 1
topaz 1>
```

Notice that the Topaz prompt always shows the number of the current session. To get a list of current GemStone sessions and the users who own them, you can execute the GemStone Smalltalk expression `System currentSessionNames`. For example:

```
topaz 1> printit
System currentSessionNames
%

session number: 2 UserId: GcUser
session number: 3 UserId: SymbolUser
session number: 4 UserId: DataCurator
session number: 5 UserId: Isaac Newton
session number: 6 UserId: Isaac Newton
session number: 7 UserId: Gottfried Leibniz
topaz 1>
```

The GcUser session (or sessions) represent the garbage collection processes that usually (though not always) operate when GemStone is active. The SymbolUser session represents the process that administers Symbols to ensure canonicity.

Keep in mind that this list includes all sessions that are currently logged into the system, not only the sessions within Topaz. The session numbers reported here do not correspond to the sequential session numbers assigned by your Topaz.

If you use the **topaz** command to invoke Topaz, you get an RPC session. With every subsequent login command you get another RPC session.

If you use the **topaz -l** command to invoke Topaz, your first session is linked. You may have only one linked session, so you need to enter a gemnetid in order to be able to log in a second session. All sessions after the first linked session will be RPC.

The messages displayed during login indicate if you have a linked or RPC session. In particular note the processId of the gem. If the processId is -1, it indicates a linked session. A positive value is the operating system processId (pid) of the gem.

```
topaz> set gemnetid gemnetobject
topaz> login
GemStone password? <password typed here but not echoed>
gci login: currSession 2 rpc gem processId 157
successful login
topaz 2> set gemnetid "
topaz 2> login
[Info]: LNK client/gem GCI levels = 830/830
[Info]: User ID: DataCurator
[Info]: Repository: gs64stone
[Info]: Session ID: 5
[Info]: GCI Client Host: <Linked>
[Info]: Page server PID: -1
[Info]: Login Time: 08/21/09 16:32:25.591 PDT
gci login: currSession 1 rpc gem processId -1
successful login
topaz 1>
```

## 1.18 Structural Access To Objects

In your GemStone Smalltalk programs, you should generally access the values stored in objects only by sending messages. During debugging, however, it's sometimes useful to be able to read an instance variable or store a value in it without sending a message. For example, if an instance variable is normally read only by a message with side effects, it won't do to examine its value during debugging by sending that message.

To allow you to "peek" and "poke" at objects without passing messages, Topaz provides the commands **object at:** and **object at: put:**.

## Examining Instance Variables with Structural Access

The command **object at:** returns the value of an instance variable within an object at some integral offset. Suppose, for example, that you had created an instance of `Animal`:

```
topaz 1> printit
UserGlobals at: #MyAnimal put: Animal new.
%
an Animal
  name          nil
  favoriteFood  nil
  habitat       nil
topaz 1> printit
MyAnimal habitat: 'water'
%
an Animal
  name          nil
  favoriteFood  nil
  habitat       water
```

The following example shows how you could use **object at:** to display the value of `MyAnimal`'s third instance variable.

```
topaz 1> object MyAnimal at: 3
water
```

You can string together **at:** parameters after **object** to descend as far as you like into the object of interest. The following example retrieves the first instance variable of `MyAnimal`'s third instance variable.

```
topaz 1> object MyAnimal at: 3 at: 1
$w
```

As far as **at:** is concerned, named, indexed, and unordered instance variables are all numbered, with named instance variables appearing first, followed by indexed instance variables, then unordered instance variables. That is, if an indexed object also had three named instance variables, the first indexable field would be addressed with **object at: 4**. Offsets into the unordered portions of NSCs are not consistent across `add:` or `remove:` commands.

## Specifying Objects

As you have seen, objects can be identified within an **object** command by global GemStone Smalltalk variable names. This is only one of several kinds of object specification acceptable in such Topaz commands as **object at:**. The others include object identity specification formats and literal object specification formats.

### Object Identity Specification Formats

*@integer*

An unsigned 64-bit decimal OOP value that denotes an object.

*integer*

A 61-bit literal SmallInteger.

*\$character*

A literal Character.

*aVariableName*

This can be either a GemStone Smalltalk variable name or a local variable created with the **define** command.

- \*\*** The object that was the result of the last execution.
- ^** The current class (as defined by the most recent **set class**, **list categoriesIn:**, **method:**, **classmethod:**, or **fileout** command).

### Literal Object Specification Formats

*'text'*

A literal String.

*#text*

A literal Symbol (no white space allowed).

*#'text'*

A quoted literal Symbol.

*float*

A Float object (C double-precision Float). The syntax for literal floating point numbers in Topaz commands is:

*[sign] digits [ . [digits] [E [sign] digits] ]*

The OOP specifications and **\*\*** (last result) are especially interesting. For example:

```
topaz 1> display oops
topaz 1> object Animal
[1337089 sz:19 cls: 150617 Animal class] Animal class
  superClass      [72193 sz:19 cls: 206081 Object class] Object class
  format          [2 sz:0 cls: 74241 SmallInteger] 0
  instVars        [26 sz:0 cls: 74241 SmallInteger] 3
  instVarNames    [1335297 sz:3 cls: 66817 Array] an Array
```

```
...
topaz 1> ! Look at first element of instVarNames array
topaz 1> object @1335297 at: 1
[1248257 sz:4 cls: 110849 Symbol] name
topaz 1> ! Look at first character of first instvarname
topaz 1> omit oops
topaz 1> object ** at: 1
$ñ
```

Note that when you look at the first element of the `instVarNames` array, you need to use the OOP returned by your own GemStone system, not `@1335297`.

## Specifying Method Selectors

When specifying a method selector in a Topaz command, you can use any of the following formats:

*text*

'*text*'

A literal String.

**#***text*

A literal Symbol (no white space allowed).

**#**'*text*'

A quoted literal Symbol.

The resulting token becomes a String object and is subsequently converted to a Symbol object if required by the command.

## 1.19 Defining Local Variables

As you saw in the last section, Topaz lets you refer to objects via their OOPs. Because long numerical OOPs are hard to remember, Topaz also provides a facility for defining local Topaz variables so that you can name those OOPs.

### Creating Variables

The following example shows the use of the Topaz **define** command to create a reasonable name for an object previously known by its OOP.

```
topaz 1> display oops
topaz 1> object Animal
[1337089 sz:19 cls: 1337601 Animal class] Animal class
  superClass [72193 sz:19 cls: 206081 Object class] Object class
  format      [2 sz:0 cls: 74241 SmallInteger] 0
  instVars    [26 sz:0 cls: 74241 SmallInteger] 3
  instVarNames[1335297 sz:3 cls: 66817 Array] an Array
...
topaz 1> define animalVars @1335297
topaz 1> omit oops
topaz 1> object animalVars at: 1
name
```

A local variable must begin with a letter or an underscore, can be up to 255 characters in length, and cannot contain white space.

If additional tokens follow **define**'s second parameter, Topaz will try to interpret them as a message to the object represented by the second parameter. For example:

```
topaz 1> define thirdvar animalVars at: 3
topaz 1> object thirdvar
habitat
```

Note that Topaz does not parse message expressions exactly as the GemStone Smalltalk compiler does; Topaz requires you to separate tokens with white space.

As the last example shows, local variables can be used in **object** commands. When used in this way, the local definition of a symbol always overrides any definition of the symbol in GemStone. For example, if "thirdvar" were defined in UserGlobals, that definition would be ignored in **object** commands.

All Topaz object specification formats (described above in “Specifying Objects”) are legal in **define** commands. For example:

```
topaz 1> define sum 1.0e1 + 500
topaz 1> define mystring 'this and that'
topaz 1> define mycharacter $z
```

## Displaying Current Variable Definitions

To see all current local variable definitions, just type **define** with no arguments:

```
topaz 1> define
Current definitions are:
mycharacter      = 142538
mystring        = 150133
sum             = 147709
thirdvar       = 114793
animalVars     = 147682
-----
ErrorCount      = nil
SourceStringClass = 74753
CurrentCategory = nil
CurrentClass    = nil
ErrorProcess    = nil
LastResult      = 147709
LastText        = nil
myUserProfile   = 13837
```

Note that **define** reports most values as OOPs rather than literals.

In this status report the user-defined local variables are listed first. The last seven items are local variables that Topaz automatically creates for you. They refer, respectively, to the number of Topaz and GemStone errors made since you started Topaz, the current category and class, the last GemStone Smalltalk execution error stack, the last execution result, the text of the last GemStone Smalltalk expression executed or compiled, and your UserProfile. You cannot modify the definitions of these predefined variables with **define**.

## Clearing Variable Definitions

To clear a definition, type **define** *aVarName* with no second argument.

For example:

```
topaz 1> define abc 'this string'
topaz 1> object abc
    this string
topaz 1> define abc
topaz 1> object abc
GemStone could not find an object named abc.
```

## 1.20 Sending Messages

Usually you'll send messages only inside methods or within **printit** commands. If you can point to an object only via a local Topaz variable or via an OOP, however, this won't work.

Therefore, Topaz provides the **send** command, which lets you send a message to an object identified by any of the means described in "Specifying Objects" on page 42. For example:

```
topaz 1> send @71425 class
a Metaclass
  superClass      a Metaclass
  format 1040
  ...
  categories      a GsMethodDictionary
  secondarySuperclasses nil
  thisClass       UndefinedObject class
```

The **send** command's first argument is an object specification identifying a receiver. That argument is followed by a message expression built almost as it would be in GemStone Smalltalk. Here's another example:

```
topaz 1> send 2 - 1
1
```

There are some differences between **send** syntax and GemStone Smalltalk expression syntax. Only one message send can be performed at a time with **send**. Cascaded messages, parenthetical messages, and the like are not recognized by this command. Also note that each item must be delimited by one or more spaces or tabs.

## 1.21 Logging Out

To log out from your current GemStone session, just type **logout**.

```
topaz 1> logout
topaz>
```

As noted above, logging out implicitly aborts your transaction.

## 1.22 Leaving Topaz

To leave Topaz and return to your host operating system, just type **exit**:

```
topaz> exit
```

If you are still logged in when you type **exit**, this will implicitly abort all your transactions and log out all active sessions.

You can use **quit**, which has the same effect as **exit**.

—  
|

# *Debugging Your GemStone Smalltalk Code*

---

Topaz can maintain up to eight simultaneous GemStone Smalltalk call stacks that provide information about the GemStone state of execution. Each call stack consists of a linked list of method or block contexts. Topaz provides debugging commands that enable you to:

- Step through execution of a method. After each step, you can examine the values of arguments, temporaries, and instance variables.
- Inspect or change the values of arguments, temporaries, and receivers in any context on the call stack, then continue execution. This means that you can find out what the system was doing at the time a soft break, a breakpoint, or an error interrupted execution.
- Set, clear, and examine GemStone Smalltalk breakpoints. When a breakpoint is encountered during normal execution, you can issue Topaz commands to explore the contexts on the stack.

This chapter introduces you to the Topaz debugging commands and provides some examples. For a detailed description of each of these commands, see Chapter 3.

## 2.1 Step Points and Breakpoints

For the purpose of determining exactly where a step will go during debugging, a GemStone Smalltalk method can be decomposed into step points. The locations of step points also determine where breakpoints can be set.

Generally, step points correspond to the message selector and, within the method, message-sends, assignments, and returns of nonatomic objects. (However, compiler optimizations may occasionally result in a different, nonintuitive step point, particularly in a loop.) The Topaz **list steps method:** command lists the source code of a given instance method and displays all step points (allowable breakpoints) in that source code. For example:

```
topaz 1> set class String
topaz 1> list steps method: includesValue:
  includesValue: aCharacter
* ^1                                                    *****

  "Returns true if the receiver contains aCharacter, false
  otherwise. The search is case-sensitive."

  <primitive: 94>

  aCharacter _validateClass: AbstractCharacter .
*           ^2                                                    *****
  ^ self includesValue: aCharacter asCharacter .
* ^5      ^4                                                    ^3          *****
```

As shown here, the position of each method step point is marked with a caret (^) and a number.

If you use the Topaz **step** command (described below) to step through this method, the first step halts execution at the beginning of the method. The second step takes you to the point where `_validateClass:` is about to be sent to `aCharacter`. Stepping again would execute that message-send and halt execution at the point where `asCharacter` is about to be sent. Another step would cause that message to be sent and then halt execution just before the message `includesValue:` is sent to `self`.

The call stack becomes active, and the debugging commands become accessible, when you execute GemStone Smalltalk code containing a breakpoint (as well as when you encounter an error). As explained earlier, you can set a breakpoint at any step point. You can use the **break** command (described below) to set method

breakpoints that halt execution at a particular step point within a method. In general, you can choose to set a method break before a message-send, an assignment, or a method return.

You can set a breakpoint on any method. Some methods, such as `Boolean>>ifTrue:` never hit the break points unless you invoke them with `perform:` or one of the **GciPerform...** functions, because sends of special selectors are optimized by the compiler.

## 2.2 Setting, Clearing, and Examining Breakpoints

You can use the **break method** and **break classmethod** commands to establish method breakpoints within your GemStone Smalltalk code:

```
break method aClassName aSelector [@ stepNumber]
break classmethod aClassName aSelector [@ stepNumber]
```

For example:

```
topaz 1> break classmethod GsFile openRead: @ 2
```

Establishes a breakpoint at step point 2 of the class method `openRead:` for `GsFile`.

```
topaz 1> set class String
topaz 1> break method ^ < @ 2
```

Establishes a breakpoint at step point 2 of the instance method “<” for the current class (`String`).

The Topaz **list breaks** command allows you to display all method breakpoints currently set in the active method context. By supplying a selector as an argument to the **list breaks** command, you can display all breakpoints set in a given instance or class method for the current class, as shown in the following example.

```

topaz 1> list breaks method: <
< 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'
"

<primitive: 28>

aCharCollection _validateClass: CharacterCollection .
*                ^2                *****
^ aCharCollection > self

```

Alternatively, you can use the **break list** command to list all currently set method or message breakpoints:

```

topaz 1> break list
1: GsFile >> nextLine @ 1
2: GsFile class >> openRead: @ 2
3: String >> < @ 2

```

In the break list, each breakpoint is identified by a break index. To disable a breakpoint, supply that break index as the single argument to the **break disable** command:

```

topaz 1> break disable 2

```

A similar command line reenables the break point:

```

topaz 1> break enable 2

```

To delete a single breakpoint, supply that break index as the argument to the **break delete** command:

```
topaz 1> break delete 2
```

To delete all currently set breakpoints, type the following command:

```
topaz 1> break delete all
```

## 2.3 Examining the GemStone Smalltalk Call Stack

When you execute the code on which you have enabled a breakpoint, execution pauses. For example, if we put a breakpoint on the setter method for Animal's instance variable #name:

```
topaz 1 > break method Animal name: @1
```

Then run this code:

```
topaz 1 > run
Animal new name: 'Dog'
%
a Breakpoint occurred (error 6005), Method breakpoint
encountered.
1 Animal >> name:                               (envId 0) @1 line 1
```

You can display all of the contexts in the active call stack by issuing the **where**, **stk** or **stack** commands with no arguments. The **where** and **stk** command display a summary call stack, with one line for each context. Use the **stack** command to display method arguments and temporaries. When using the **stack** command, the volume of output displayed is controlled by the current **level** setting.

This is an example of the **where** summary:

```
topaz 1> where
==> 1 Animal >> name:                               (envId 0) @1 line 1
2 Executed Code                                     @3 line 1
3 GsNMethod class >> _gsReturnToC                 (envId 0) @1 line 1
```

With display oops active, the **where** command provides more detail:

```
topaz 1> display oops
topaz 1> where
==> 1 Animal >> name:                               (envId 0) @1 line 1 [methId 27134209]
2 Executed Code                                     @3 line 1 [methId 27128321]
3 GsNMethod class >> _gsReturnToC                 (envId 0) @1 line 1 [methId 4520961]
```

```
[GsProcess 27133441]
```

The display for **stack**, for **level 0** (the default), provides additional information about the instance and temporary variable names and values:

```
topaz 1> stack
==> 1 Animal >> name:                (envId 0) @1 line 1 [methId 27134209]
    receiver [27125505 sz:3 cls: 27141121 Animal] anAnimal
    newValue [27124737 sz:3 cls: 74753 String] Dog
2 Executed Code                        @3 line 1 [methId 27128321]
    receiver [20 sz:0 cls: 76289 UndefinedObject] nil
3 GsNMethod class >> _gsReturnToC    (envId 0) @1 line 1 [methId 4520961]
    receiver [20 sz:0 cls: 76289 UndefinedObject] nil
```

With **level 1**, more detail is displayed. For example, with **omit oops**:

```
topaz 1> omit oops
topaz 1> level 1
topaz 1> stack
==> 1 Animal >> name:                (envId 0) @1 line 1
    receiver an Animal
        name                nil
        favoriteFood        nil
        habitat              nil
    newValue Dog
2 Executed Code                @3 line 1
    receiver nil
3 GsNMethod class >> _gsReturnToC    (envId 0) @1 line 1
    receiver nil
```

The display of each context includes:

- an indicator of the active context, a preceding ==>
- the level number of the context;
- the OOP of the GsMethod (if **display oops** is active)
- the class of the method invoked
- the selector of the method
- the current step point within the method, indicated by *@anInteger*
- line number of the step point within the source code

- for the stack command, the receiver, parameters and temporaries for this context (including method temporaries and OOPs, if **display oops** is active).

The display is governed by the setting of other Topaz commands such as **limit**, **level**, and **display** or **omit**.

## Proceeding After a Breakpoint

When GemStone Smalltalk encounters a breakpoint during normal execution, Topaz halts and waits for your reply. Topaz provides commands for continuing execution, and for stepping into and over message-sends.

### **continue**

Tells GemStone Smalltalk to continue execution from the context at the top of the stack, if possible. If execution halts because of an authorization error, for example, then the virtual machine can't continue. As an option, the **continue** command can replace the value on the top of the stack with another object before it attempts to continue execution.

### **step over**

Tells GemStone Smalltalk to advance execution to the next step point (message-send, assignment, etc.) in the active context or its caller, and halt. The active context is indicated by the ==> in the stack; it is the context specified by the last **frame**, **up**, **down** or another command. Initially it is the top of the stack (the first context in the list).

### **step into**

Tells GemStone Smalltalk to advance execution to the next step point (message-send, assignment, etc.) and halt. If the current step point is a message-send, then execution will halt at the first step point within the method invoked by that message-send.

Notice how this differs from **step over**; if the next message in the context contains step points itself, execution halts at the first of those step points. That is, the virtual machine "steps into" the new method instead of silently executing that method's instructions and halting after the method has completed. The next **step over** command will then take place within the context of the new method.

## Examining and Modifying Temporaries and Arguments

The Topaz **temporary** command lets you examine or modify the values of temporaries in the active context. If, for example, the method under inspection had

a temporary variable named `count`, that currently had a value of 5, you could obtain its value by typing **temporary** and the variable name:

```
topaz 1> temporary count
5
```

Similarly, you can use the **temporary** command to assign a new value to a temporary variable:

```
topaz 1> temporary count 8
```

For example, the following code sets a breakpoint, executes code, views and updates the value of a temporary variable, then continues execution to return the results of the code; which has been changed during debugging.

```
topaz 1> break classmethod String withAll:
topaz 1> run
String withAll: 'abc'
%
a Breakpoint occurred (error 6005), Method breakpoint
encountered.
1 String class >> withAll:                (envId 0) @1 line 1
topaz 1> stack
==> 1 String class >> withAll:            (envId 0) @1 line 1
    receiver String
    aString abc
2 Executed Code                            @2 line 1
    receiver nil
3 GsNMethod class >> _gsReturnToC         (envId 0) @1 line 1
    receiver nil
topaz 1> temporary
    aString abc
topaz 1> temporary aString 'xyz'
topaz 1> stack
==> 1 String class >> withAll:            (envId 0) @1 line 1
    receiver String
    aString xyz
2 Executed Code                            @2 line 1
    receiver nil
3 GsNMethod class >> _gsReturnToC         (envId 0) @1 line 1
    receiver nil
topaz 1> continue
xyz
```

## Select a Context for Examination and Debugging

The Topaz commands **frame**, **up**, and **down**, as well as **stack up**, **stack down**, and **stack scope**, let you redefine the active context (used by the **temporary**, **stack**, and **list** commands) within the current call stack. Recall the stack we examined earlier:

```
topaz 1> stack
==> 1 Animal >> name:                (envId 0) @1 line 1
      receiver anAnimal
      aString Dog
2 Executed Code                        @3 line 1
      receiver nil
3 GsNMethod class >> _gsReturnToC     (envId 0) @1 line 1
      receiver nil
```

The active context is indicated by ==>. You can also show the active context by using the **frame** command with no arguments:

```
topaz 1> frame
1 Animal >> name:                (envId 0) @1 line 1
      receiver anAnimal
      aString Dog
```

The following command selects the caller of this context as the new active context:

```
topaz 1> frame 2
2 Executed Code                        @3 line 1
      receiver nil
```

Now confirm that Topaz redefined the active context:

```
topaz 1> where
1 Animal >> name:                (envId 0) @1 line 1
==> 2 Executed Code                @3 line 1
3 GsNMethod class >> _gsReturnToC (envId 0) @1 line 1
```

You can also use **up** and **down** commands to make a different frame the active context.

—  
|

# Command Dictionary

---

This chapter provides descriptions of each Topaz command, in alphabetical order.

## Command Syntax

Most Topaz commands can be abbreviated to uniqueness. For example, **set password:** can be shortened to **set pass**. Exceptions to this rule are a few commands whose actions can affect the success or failure of your current transaction and, thus, the integrity of your data: **abort**, **begin**, **commit**, **exit**, **logout**, **removeallmethods**, **removeallclassmethods**, **output push**, and **output pop**.

Topaz commands are case-insensitive. **Time**, **TIME**, and **time** are understood by Topaz as the same command. However, arguments you supply to Topaz commands may be subject to case-sensitivity constraints. For example, the commands **category: animal** and **category: Animal** specify two different categories, since GemStone Smalltalk is case-sensitive. The same is true of UNIX path names, user names, and passwords.

Objects passed as arguments to Topaz commands can usually be specified using the formats described in “Specifying Objects” on page 42.

Command lines can have as many as 511 characters. You can stop a command at any time by typing **Ctrl-C**. Topaz may take a moment or two before responding.

## ABORT

Aborts the current GemStone transaction. Your local variables (created with the **define** command) may no longer have valid definitions after you abort.

If your session is outside a transaction, use **abort** to give you a new view of the repository.

Although you can abbreviate most other Topaz commands and parameter names, **abort** must be typed in full.

## BEGIN

Begins a GemStone transaction when your session is outside a transaction.

When you have ended your transaction by invoking the GemStone Smalltalk method

```
System transactionMode: #manualBegin
```

use **begin** to start a new transaction. For more information, see the protocol for System Class. Although you can abbreviate most other Topaz commands and parameter names, **begin** must be typed in full.

## BREAK *aSubCommand*

Establishes (or displays) a method breakpoint within your GemStone Smalltalk code. Subcommands are **method**, **classmethod**, **list**, **enable**, **disable**, and **delete**. For more information about breakpoints, see Chapter 2, “Debugging Your GemStone Smalltalk Code.”

### Method Breakpoints

You can set method breakpoints within an instance method at step points: assignments, message sends, or method returns. Use the **list steps** command to display all valid step points for a method.

In each of the following commands, the optional argument *anInt* specifies the step point within that method where the break is to occur. If you do not specify *anInt*, the breakpoint is established at step 1 of the method.

You may not set method breakpoints in any method whose sole function is to perform any of the following actions: return self, return nil, return true, return false, return or update the value of an instance variable, return the value of a literal, or return the value of a literal variable (that is, a class variable, a pool variable, or a variable defined in your symbol list).

You may supply the class name parameter in these four formats:

*@integer*

An unsigned 64-bit decimal OOP value that denotes an object.

*aVariableName*

This can be either a GemStone Smalltalk variable name or a local variable created with the **define** command.

**\*\*** The object that was the result of the last execution.

**^** The current class (as defined by the most recent **set class:**, **list categoriesin:**, **method:**, **classmethod:**, **removeallmethods**, **removeallclassmethods**, or **fileout class:** command).

**break method** *aClassName selectorSpec* [*@ anInt*]

Establishes a method breakpoint on the given instance method.

**break classmethod** *aClassName selectorSpec* [*@ anInt*]

Establishes a method breakpoint on the given class method.

**break method** **^** *selectorSpec* [*@ anInt*]

Establishes a method breakpoint on the given instance method for the current class.

**break classmethod** ^ *selectorSpec* [*@ anInt*]

Establishes a method breakpoint on the given class method for the current class.

**break method** *@anObjectSpec selectorSpec*

Establishes a method breakpoint at step point 1 of the given instance method for the class with the specified objectId.

**break method** *@anObjectSpec*

Establishes a method breakpoint at step point 1 of the GsNMethod with the specified objectId.

**break** *@anInt*

Establishes a method breakpoint at the specified step point of the method selected by the previous **down**, **frame**, **lookup**, **list**, or **up** command.

## Displaying Breakpoints

**break list**

Lists all currently set breakpoints. In the display, each breakpoint is identified by a break index for subsequent use in **break disable**, **break enable**, and **break delete** commands.

## Disabling and Enabling Breakpoints

**break disable** *anIndex*

Disables the breakpoint identified by *anIndex* in the **break list** command.

**break disable all**

Disables all currently set breakpoints.

**break enable** *anIndex*

Reenables the breakpoint identified by *anIndex* in the **break list** command.

**break enable all**

Reenables all disabled breakpoints.

## Deleting Breakpoints

**break delete** *anIndex*

Deletes the breakpoint identified by *anIndex* in the **break list** command.

**break delete all**

Deletes all currently set breakpoints.

## Examples

```
topaz 1> break method GsFile nextLine
```

Establishes a breakpoint at step point 1 of the instance method `nextLine` for `GsFile`.

```
topaz 1> break classmethod GsFile openRead: @ 2
```

Establishes a breakpoint at step point 2 of the class method `openRead:` for `GsFile`.

```
topaz 1> set class String
```

```
topaz 1> break method ^ < @ 2
```

Establishes a breakpoint at step point 2 of the instance method “<” for the current class (`String`).

```
topaz 1> break list
```

```
1: GsFile >> nextLine @ 1
```

```
2: GsFile class >> openRead: @ 2
```

```
3: String >> < @ 2
```

```
topaz 1> break disable 2
```

```
topaz 1> break list
```

```
1: GsFile >> nextLine @ 1
```

```
2: GsFile class >> openRead: @ 2 (disabled)
```

```
3: String >> < @ 2
```

```
topaz 1> break enable 2
```

```
topaz 1> break list
```

```
1: GsFile >> nextLine @ 1
```

```
2: GsFile class >> openRead: @ 2
```

```
3: String >> < @ 2
```

```
topaz 1> break delete 1
```

```
topaz 1> break list
```

```
2: GsFile class >> openRead: @ 2
```

```
3: String >> < @ 2
```

```
topaz 1> break delete all
```

```
topaz 1> break list
```

```
No breaks set
```

## CATEGORY: *aCategoryName*

Sets the current category, the category for subsequent method compilations. If you try to compile a method without first selecting a category, the new method is inserted in the default category "as yet unspecified." This command has the same effect as the **set category:** command.

If the category you name doesn't already exist, Topaz creates it when you first compile a method. If you wish to include spaces in the category name you specify, enclose the category name in single quotes.

Specifying a new class with **set class** does not change your category. However, when you **edit** or **fileout** a method, that method's category becomes the current category.

The current category is cleared by the **logout**, **login**, and **set session** commands.

```
topaz 1> category: Accessing
topaz 1> category: 'Public Methods'
```

## CLASSMETHOD[: *aClassName*]

Compiles a class method for the class whose name is given as a parameter. The class of the method you compile is automatically selected as the current class. If you don't supply a class name, the method is compiled for the current class (as defined by the most recent **set class;** **list categoriesin;** **method;** **classmethod;** **removeAllMethods**, **removeAllClassMethods**, or **fileout class:** command).

Text of the method should follow this command on subsequent lines. The method text is terminated by the first line that contains a % character in column 1. For example:

```
topaz 1> classmethod: Animal  
returnAString  
    ^String new  
%
```

Topaz sends the method's text to GemStone for compilation and inclusion in the current category of the specified class. If you haven't yet selected a current category, the new method is inserted in the default category "as yet unspecified."

## COMMIT

Ends the current GemStone transaction and stores your changes in the repository. Although you can abbreviate most other Topaz commands and parameter names, **commit** must be typed in full.

## CONTINUE [*anObjectSpec*]

Attempts to continue GemStone Smalltalk execution on the active call stack after encountering a breakpoint, a `pause` message, or a user-defined error. The call stack becomes active, and the **continue** command becomes accessible, when you execute GemStone Smalltalk code containing a breakpoint.

### **continue**

Attempts to continue execution. (The `continue` command may be abbreviated as `c`.)

### **continue** *anObjectSpec*

Replaces the value on the top of the stack with *anObjectSpec* and attempts to continue execution.

The argument *anObjectSpec* can be specified using any of the formats described in “Specifying Objects” on page 42.

For more information about breakpoints, see the discussion of the **break** command on page 62, or see Chapter 2, “Debugging Your GemStone Smalltalk Code”.

For information about replacing the value on the top of the stack, see the **GciContinueWith** function in the *GemBuilder for C* manual.

For information about Object’s `pause` method, see the method comments for `Object>>pause`.

For information about user-defined errors, see the discussion of error-handling in the *GemStone/S 64 Bit Programming Guide*. User manuals for the GemStone interfaces, such as *GemBuilder for Smalltalk*, also contain discussions of error-handling.

## **DEFINE [*aVarName* [*anObjectSpec* [*aSelectorOrArg*]...]]**

Defines local Topaz variables that allow you to refer to objects in commands such as **send** and **object**.

All Topaz object specification formats (as described in “Specifying Objects” on page 42) are legal in **define** commands.

### **define**

Lists all current local variable definitions.

### **define** *aVarName*

Deletes the definition of the variable *aVarName*.

### **define** *aVarName anObjectSpec aSelectorOrArg ...*

Sends a message to the object specified by *anObjectSpec*, and saves the result as a local variable with the name *aVarName*. The variable name *aVarName* must begin with a letter (a..z) or an underscore, can be up to 255 characters in length, and cannot contain white space.

```
topaz 1> define CurrentSessions System currentSessionNames  
topaz 1> define UserId myUserProfile userId
```

Topaz tries to interpret all command line tokens following *anObjectSpec* as a message to the specified object.

## DISASSEM [*aClassParameter*] *aParamValue*

The **disassem** command allows you to disassemble the specified GsNMethod, displaying the assembly code instructions.

The **disassem** command is intended for use in a linked (**topaz -l**) session only. If the session is remote, the output goes to stdout of the remote Gem, which is the gem log.

**disassem** *@anOop*

Disassemble the method or code object with the specified oop.

**disassem method:** *aSelectorSpec*

Disassemble the specified instance method for the class previous set by the **set class** command.

**disassem classmethod:** *aSelectorSpec*

Disassemble the specified class method for the class previous set by the **set class** command.

## DISPLAY *aDisplayFeature*

The **display** and **omit** commands control the display of instance variable names, hexadecimal byte values, and OOPs (object-oriented pointers). The **display** command turns on these display attributes, and the **omit** command turns them off.

### **display oops**

For each object, displays a header containing the object's OOP (a 64-bit unsigned integer), the object's size (the sum of its named, indexed, and unordered instance variable fields), and the OOP of the object's class.

### **display bytes**

When displaying string objects, includes the hexadecimal value of each byte.

### **display errorcheck**

Allows Topaz programs to automatically record the results of error checking. Using this command creates the `./topazerrors.log` file or opens the file to append to it, if it already exists.

As long as **display errorcheck** is set, every time `ErrorCount` is incremented, a summary of the error is added to `topazerrors.log`. The summary includes the line number in the Topaz output file, if possible. If the only output file open is `stdout`, then line numbers are not available. To close the `topazerrors.log` file, use the **omit errorcheck** command. Subsequent results are not recorded.

### **display names**

For each of an object's named instance variables, displays the instance variable name along with its value. (This is the default condition.) To turn off this display, use the **omit names** command.

When instance variable name display is off, named instance variables appear as `i1`, `i2`, `i3`, and so on.

### **display resultCheck**

Allows Topaz programs to check input values and record the results. This command creates the `./topazerrors.log` file or opens the file to append to it, if it already exists. Specifying **display resultCheck** is equivalent to setting **expectvalue true**, except that it affects the behavior of all **run** and **printit** commands, not only the next one.

As long as **display resultCheck** is set, every time `ErrorCount` is incremented, a summary of the error is added to `topazerrors.log`. This includes the line number in the Topaz output file, if possible. If the only output file open is `stdout`, then line numbers are not available. To close the file, use the **omit resultCheck** command. Then the results of a successful **run** or

**printit** command will no longer be checked, unless an **expectvalue** command precedes the **printit** command.

**display pauseonerror**

When an error occurs, if Topaz is receiving input from a terminal, displays the message:

```
Execution has been suspended by a "pause" message.  
Topaz pausing after error, type <return> to continue,  
ctl-C to quit ?
```

and waits for the user to press the **Return** key to continue execution. Pressing **Ctrl-C** ends the pause and stops the processing of input files altogether.

If **display resultCheck** is also set, then Topaz only pauses when the result or error is contrary to the current **resultCheck**, **expectvalue**, and **expecterror** settings.

When **display pauseonerror** is set, the **status** command output includes:

```
display interactive pause on errors
```

Use **omit pauseonerror** to cancel this mode.

**display classoops**

If **display oops** is set, enables the display of OOPs of classes along with class names in object display. Also causes the OOPs of classes to be printed by stack display and method lookup commands, and enables the printing of evaluation temporary objects in stack frame printouts from the **frame** command.

**display lineditor**

Enables the use of the Topaz line editor, using the open source linenoise library. (This is the default condition.) To disable use of the line editor, use the **omit lineditor** command. Not available on Windows.

**display pushonly**

Enables the effect of the **only** keyword in an **object push** command. To disable this effect, use the **omit pushonly** command.

**display zerobased**

Shows offsets of instance variables as zero-based when displaying objects. (By default, offsets are one-based.) To show offsets as one-based, use the **omit zerobased** command.

## DOIT

Sends the text following the **doit** command to the object server for execution and displays the OOP of the resulting object. If there is an error in your code, Topaz displays an error message instead of a legitimate result. GemStone Smalltalk text is terminated by the first line that contains a % in column 1. For example:

```
topaz 1> doit
2 + 1
%
result oop is 26
```

The text executed between the **doit** and the terminating % can be any legal GemStone Smalltalk code, and follows all the behavior documented in the *GemStone/S 64 Bit Programming Guide*.

If the configuration parameter GEM\_NATIVE\_CODE\_ENABLED is set to FALSE, or if any breakpoints are set, execution defaults to interpreted mode. Otherwise, execution defaults to using native mode.

- For details about GemStone configuration parameters, see the *System Administration Guide for GemStone/S 64 Bit*.

Note that **doit** always displays results at level 0, regardless of the current display level setting (page 103). The **doit** command does not alter the current level setting.

## DOWN [*anInteger*]

Moves the active context down within the currently selected call stack, and displays the activation selected as a result. The optional argument *anInteger* specifies how many activations to move down. If no argument is supplied, the scope will go down one activation. See also **stack down** on page 153.

The context displayed includes parameters and temporaries for the context, unlike the results displayed by **stack down**.

```
topaz 1> where
1 AbstractException >> _signalWith:      (envId 0) @5 line 25
2 AbstractException >> signal            (envId 0) @2 line 47
3 Number >> _errorDivideByZero          (envId 0) @6 line 7
==> 4 SmallInteger >> /                  (envId 0) @6 line 7
5 [] in Executed Code                    (envId 0) @2 line 1
6 Collection >> do:                       (envId 0) @5 line 10
7 Executed Code                           @2 line 1
8 GsNMethod class >> _gsReturnToC       (envId 0) @1 line 1
topaz 1> down
3 Number >> _errorDivideByZero           (envId 0) @6
line 7
    receiver 1
topaz 1> where
1 AbstractException >> _signalWith:      (envId 0) @5 line 25
2 AbstractException >> signal            (envId 0) @2 line 47
==> 3 Number >> _errorDivideByZero       (envId 0) @6 line 7
4 SmallInteger >> /                      (envId 0) @6 line 7
5 [] in Executed Code                    (envId 0) @2 line 1
6 Collection >> do:                       (envId 0) @5 line 10
7 Executed Code                           @2 line 1
8 GsNMethod class >> _gsReturnToC       (envId 0) @1 line 1
```

**DOWNR** [*anInteger*]

**DOWNRB** [*anInteger*]

These commands are used with Ruby applications, but not with Smalltalk applications.

## EDIT *aSubCommandOrSelector* [*aSelector*]

Allows you to edit GemStone Smalltalk source code. You can create or modify methods or blocks of code to be executed. You can also edit the text of the last **run**, **printit**, **doit**, **method:**, or **classmethod:** command.

Before you can use this command, you must first establish the name of the host operating system editor you wish to use. You can do this by setting the host environment variable `EDITOR` or by invoking the Topaz **set editorname** command interactively or in your Topaz initialization file.

Do not use the **edit** command for batch processing. Instead, use the **method:** and **classmethod:** commands to create methods in batch processes, and the **run**, **printit** or **doit** commands to execute blocks of code in batch.

If you supply any parameter to **edit**, other than one of its subcommands, Topaz assumes that you are naming an existing instance method to be edited.

## Creating or Modifying Blocks of GemStone Smalltalk Code

### **edit last**

Allows you to edit the text of the last **run**, **printit**, **doit**, **method:**, or **classmethod:** command. (You can inspect that text before you edit by issuing the Topaz command **object LastText**.) Topaz opens, as a subprocess, the editor that you've selected. When you exit the editor, Topaz saves the edited text in its temporary file and asks you whether you'd like to compile and execute the altered code. If you tell Topaz to execute the code, it effectively reissues your **run** or **printit** command with the new text.

### **edit new text**

Allows you to create a new block of GemStone Smalltalk code for compilation and execution. This is similar to **edit last**, but with a new text object.

## Creating or Modifying GemStone Smalltalk Methods

### **edit new**

If you type **edit new** with no additional keywords, Topaz assumes that you want to create a new instance method for the current class.

### **edit new method**

Allows you to create a new instance method for the current class and category. Before you can use this command, you must first use **set class** to select the current class. If you haven't yet selected a current category, the new method is inserted in the default category, "`as yet unspecified`."

**edit new classmethod**

Allows you to create a new class method for the current class and category. Before you can use this command, you must first use **set class** to select the current class. If you haven't yet selected a current category, the new method is inserted in the default category, "as yet unspecified."

**edit selectorSpec****edit method:** *selectorSpec*

Allows you to edit the source code of an existing instance method. Before you can use this command, you must first use **set class** to select the current class. The category of the method you edit is automatically selected as the current category. For example:

```
topaz 1> set class Animal
topaz 1> edit habitat
```

edits the instance method in class Animal whose selector is habitat.

**edit classmethod:** *selectorSpec*

Allows you to edit the source code of an existing class method. Before you can use this command, you must first use **set class** to select the current class. The category of the method you edit is automatically selected as the current category.

## ERRORCOUNT

Displays the Topaz `errorCount` variable, which stores the number of errors made in all sessions since you started Topaz. This includes GemStone Smalltalk errors generated by compiling or a **run** or **printit** command, as well as errors in Topaz command processing.

If **expecterror** is specified immediately before a compile or execute command (**run**, **printit**, **doit**, **method:**, **classmethod:**, **send**, or **commit**) and the expected error occurs during the compile or execute, the `ErrorCount` is not incremented. The `ErrorCount` is not reset by **login**, **commit**, **abort**, or **logout**.

You can use the **errorCount** command at the `topaz>` prompt before you log in, as well as after login. It is equivalent to

```
topaz 1> object ErrorCount
```

except that **errorCount** does not require a valid session.

## **EXIT** [*aSmallInt* / *anObjectSpec*]

Leaves Topaz, returning to the parent process or operating system. If you are still logged in to GemStone when you type **exit**, this aborts your transactions and logs out all active sessions. Although you can abbreviate most other Topaz commands and parameter names, **exit** must be typed in full.

You can include an argument (a `SmallInteger`, or an object specification that resolves to a `SmallInteger`) to specify an explicit `exitStatus` for the Topaz process. If you do not specify an argument, the `exitStatus` will be either 0 (no errors occurred during Topaz execution) or 1 (there was a GCI error or the Topaz `errorCount` was nonzero).

## EXITIFNOERROR

If there have been no errors – either GemStone Smalltalk errors or Topaz command processing errors – in any session since you started Topaz, this command has the same effect as **exit 0** (see page 79). Otherwise, this command has no effect.

## EXPECTBUG *bugNumber*

**value** *resultSpec* [ *integer* ] |

**error** *errCategory* *errNumCls* [ *resultSpec* [ *resultSpec* ]..]

Specifies that the result of the following execution results in the specified answer (either a value or an error). If the expected result occurs, Topaz prints a confirmation message and increments the error count.

The **expectbug** command is intended for use in self-checking scripts to verify the existence of a known error. Only one **expectbug** command (at most) can be in effect during a given execution. Topaz honors the last **expectbug** command issued before the execution occurs. **Expectbug** can be used in conjunction with the **expecterror** and **expectvalue** commands – an **expectbug** command does not count against the maximum of five such **expecterror** and **expectvalue** commands permitted.

*bugNumber* is a parameter identifying the bug or behavior you expect to see. In most cases this would be a number, but it can equally well be a character string. (If it contains white space, enclose the string in single quotes.) The parameter is included in the confirmation message.

*resultSpec* is specified as in the **expectvalue** command (page 85).

*errCategory* and *errNumCls*

are specified as in the **expecterror** command (page 82).

For example, suppose you know that the '\*' operator has been reimplemented in a way that returns the erroneous answer '5' for the expression '2 \* 3'. You can use the **expectbug** command in a script to verify that the bug is present:

```
topaz 1> expectbug 123 value 5
topaz 1> printit
2 * 3
%
5
BUG EXPECTED: BUG NUMBER 123
topaz 1>
```

If the expected bug does not occur, Topaz checks for an **expecterror** or **expectvalue** command that matches the answer received. If it finds a match, Topaz displays a "FIXED BUG" message. If not, the error is reported in the same way the **expecterror** or **expectvalue** command would report it ("ERROR: WRONG VALUE" for example). If no **expecterror** or **expectvalue** commands are in effect, execution proceeds without comment.

## EXPECTERROR *anErrorCategory anErrorNumCls* [*anErrorArg*[*anErrorArg*] ...]

Indicates that the next compilation or execution is expected to result in the specified error. If the expected result occurs, Topaz reports the error in the conventional manner but does not increment its error count and allows execution to proceed without further action or comment.

If the execution returns a result other than the expected error (including unexpected success), Topaz increments the error count and invokes any **iferror** actions that have been established.

Up to five **expecterror** or **expectvalue** commands may precede an execution command. If the result of the execution satisfies any one of them, the error count variable is not incremented. This mechanism allows you to build self-checking scripts to check for errors that can't be caught with GemStone Smalltalk exception handlers.

**Expecterror** must be reset for each command; it is only checked against a single return value. **Expecterror** is normally used before the commands **run**, **printit**, **doit**, **method:**, **classmethod:**, **commit**, and **send**.

*anErrorCategory* must be a Topaz object specification that evaluates to the object identifier of an error category.

*anErrorNumCls* must be a Topaz object specification that evaluates to a `SmallInteger`, or to the object identifier of a subclass of `AbstractException`.

All Topaz object specification formats (as described in "Specifying Objects" on page 42) are legal in **expecterror** commands. In addition, this command takes two more formats that allow you to specify instances of classes as error arguments:

**%className** An instance of the class *className*.

**/className** An instance of the class *className* or an instance of any of its subclasses. (In other words, an instance of a 'kind of' *className*.)

If *anErrorArg* is a literal object specification (*literalObjectSpec*), Topaz regards it as matching the result if the two are equal (=).

If *anErrorArg* is an object specification (*ObjectSpec*), Topaz regards it as matching the result if the two are identical (==).

If you care about the number of error arguments, include that many *anErrorArg* tokens on the command line. The error count is incremented if the actual error contains fewer arguments than were specified in the **expecterror** command; it is

not incremented if the error contains more arguments than specified. If you don't care about the class of the error arguments, specify /Object for each one.

The following example shows an **expecterror** command followed by the expected error. Note that although the error is reported, the error count is not incremented, nor is any additional annotation returned.

```
topaz 1> errorcount
0
topaz 1> expecterror GemStoneError MessageNotUnderstood
topaz 1> printit
1 x
%
ERROR 2010 , a MessageNotUnderstood occurred (error
2010), a SmallInteger does not understand #'x'
(MessageNotUnderstood)
topaz 1> errorcount
0
topaz 1>
```

If execution returns unanticipated results, Topaz prints a message (in this example, "ERROR: WRONG ERROR CATEGORY/NUMBER"), then invokes the actions established by the **iferror** command (in this example, a stack dump) and bumps the error count:

```
topaz 1> errorcount
0
topaz 1> iferror where
topaz 1> expecterror GemStoneError MessageNotUnderstood
topaz 1> printit
1 / 0
%
ERROR 2026 , a ZeroDivide occurred (error 2026),
reason:numErrIntDivisionByZero, An attempt was made to
divide nil by zero. (ZeroDivide)
topaz > exec iferr 0 : where
==> 1 AbstractException >> _signalWith: (envId 0) @5 line 25
2 AbstractException >> signal (envId 0) @2 line 47
3 Number >> _errorDivideByZero (envId 0) @6 line 7
4 SmallInteger >> / (envId 0) @6 line 7
5 Executed Code @2 line 1
6 GsNMethod class >> _gsReturnToC (envId 0) @1 line 1
topaz 1> errorcount
1
topaz 1>
```

## EXPECTVALUE *anObjectSpec* [*anInt*]

Indicates that the result of the following compilation or execution is expected to be a specified value, denoted by *anObjectSpec*. If it is not, the error count is incremented. Up to five **expectvalue** or **expecterror** commands may precede an execution command. If the result of the execution satisfies any one of them, the error count variable is not incremented.

**Expectvalue** must be reset for each command; it is only checked against a single return value. **Expectvalue** is normally used before the commands **run**, **printit**, **doit**, **method:**, **classmethod:**, **commit**, and **send**.

All Topaz object specification formats (as described in “Specifying Objects” on page 42) are legal in **expectvalue** commands. In addition, this command takes further formats that allow you to specify instances of classes:

*%className*

An instance of the class *className*.

*%@OOPOfClass*

An instance of the class that has the OOP *OOPOfClass*.

*/className*

An instance of the class *className* or an instance of any of its subclasses. (In other words, an instance of a ‘kind of’ *className*.)

*/@OOPOfClass*

An instance of the class that has the OOP *OOPOfClass*, or an instance of any of its subclasses.

If the argument is a literal object specification (*literalObjectSpec*), Topaz regards it as matching the result if the two are equal (=).

If the argument is an object specification (*ObjectSpec*), Topaz regards it as matching the result if the two are identical (==).

If the *anInt* argument is present, the result of sending the method `size` to the result of the following execution must be the integer *anInt*.

The **commit** command has an internal result of true for success and false for failure. All other Topaz commands have an internal result of true for success and @0 for failure.

The following example uses **expectvalue** to test that the result of the **printit** command is a **SmallInteger**. The expected result is returned, so execution proceeds without comment:

```
topaz 1> expectvalue %SmallInteger
topaz 1> printit
2 * 5
%
10
topaz 1>
```

If execution returns unanticipated results, Topaz prints a message (in this example, "ERROR: WRONG VALUE"), then invokes the actions established by the **iferror** command (in this example, a stack dump) and bumps the error count:

```
topaz 1> iferror stack
topaz 1> expectvalue %SmallInteger
topaz 1> errorcount
0
%
topaz 1> expectvalue %SmallInteger
topaz 1> printit
2 * 5.5
%
1.1000000000000000E+01
ERROR: WRONG VALUE
Now executing the following command saved from "iferror":
    stack
Stack is not active
topaz 1> errorcount
1
topaz 1>
```

## FILEOUT *aSubCommandOrSelector* [TOFILE: *aFileName*]

Writes out class-related information in a format that can be fed back into Topaz with the **input** command. To send this information to a file, use the **toFile:** keyword. For example:

```
topaz 1> fileout class: Object toFile: object.gs
```

If you specify a host environment name such as `$HOME/foo.bar` as the output file, Topaz expands that name to the full filename. If the output file does not include an explicit path specification, Topaz writes to the named file in the directory where you started Topaz.

**fileout class:** [*aClassName*]

Writes out the class definition and all the method categories and their methods. To write out the definition of the current class, type:

```
topaz 1> fileout class: ^
```

If you omit the class name parameter, the current class is written out.

The class that you file out becomes the current class for subsequent Topaz commands.

**fileout category:** *aCategoryName*

Writes out all the methods contained in the named category for the current class.

**fileout classcategory:** *aCategoryName*

Writes out all the class methods contained in the named category for the current class.

**fileout classmethod:** *selectorSpec*

Writes out the specified class method (as defined for the current class). The category of that method will automatically be selected as the current category.

**fileout method:** *selectorSpec*

Writes out the specified method (as defined for the current class). The category of that method will automatically be selected as the current category.

**fileout** *selectorSpec*

Writes out the specified method (as defined for the current class). You may use this form of the **fileout** command (that is, you may omit the **method:** keyword) only if the selector that you specify does not conflict with one of the other **fileout** keywords. For example, to file out a method named `category:`, you would need to explicitly include the **method:** keyword as shown here.

```
topaz 1> fileout method: category:
```

## **FR\_CLS** [*anInteger*]

Similar to the `frame` command (page 89), but also displays OOPs of classes along with class names in the specified stack frames.

## FRAME [*anInteger*]

Moves the active context to the context specified by *anInteger*, within the currently active call stack, and displays the activation selected as a result. The display includes parameters and temporaries.

If no argument is supplied, displays the current active context.

See also **stack scope** on page 153, the up command on page 169 and the down command on page 74.

For example:

```
topaz 1> where
==> 1 AbstractException >> _signalWith: (envId 0) @5 line 25
2 AbstractException >> signal (envId 0) @2 line 47
3 Number >> _errorDivideByZero (envId 0) @6 line 7
4 SmallInteger >> / (envId 0) @6 line 7
5 [] in Executed Code (envId 0) @2 line 1
6 Collection >> do: (envId 0) @5 line 10
7 Executed Code @2 line 1
8 GsNMethod class >> _gsReturnToC (envId 0) @1 line 1
topaz 1> frame 4
4 SmallInteger >> / (envId 0) @6 line 7
    receiver 1
    aNumber 0
topaz 1> where
1 AbstractException >> _signalWith: (envId 0) @5 line 25
2 AbstractException >> signal (envId 0) @2 line 47
3 Number >> _errorDivideByZero (envId 0) @6 line 7
==> 4 SmallInteger >> / (envId 0) @6 line 7
5 [] in Executed Code (envId 0) @2 line 1
6 Collection >> do: (envId 0) @5 line 10
7 Executed Code @2 line 1
8 GsNMethod class >> _gsReturnToC (envId 0) @1 line 1
```

## **GCITRACE** *aFileName*

Turns GCI tracing on. Subsequent GCI calls are logged to the file *aFileName*. If *aFileName* is "" (empty string), then turns GCI tracing off.

—  
|

## HELP [*aTopicName*]

Invokes a hierarchically-organized help facility that can provide information about all Topaz commands. Enter ? at a help prompt for a list of topics available at that level of the hierarchy. Help topics can be abbreviated to uniqueness.

To display help text for **fileout**:

```
topaz 1> help fileout
```

To display help text for **last**:

```
topaz 1> help edit last
```

Press *Return* at a help prompt to go up a level in the hierarchy until you exit the help facility.

## **HIERARCHY** [*aClassName*]

Prints the class hierarchy up to Object for the specified class. If you don't specify a class, Topaz prints the hierarchy for the current class.

## HISTORY [*anInteger*]

Displays the specified number of recently executed commands, as listed in the Topaz line editor history. Has no effect if the line editor is not enabled. (Not available on Windows.)

The **set history** command (page 144) establishes the maximum number of command lines to retain in the Topaz line editor history.

## IFERR *bufferNumber* [*aTopazCommandLine*]

The **iferr** command works whenever an error is reported and the ErrorCount variable is incremented.

This command saves *aTopazCommandLine* in the post-error buffer specified by *bufferNumber* as an unparsed Topaz command line. There are 10 buffers; *bufferNumber* must be a number between 1 and 10, inclusive.

The post-error buffer commands apply under any of the following conditions:

- an error occurs (other than one matching an **expecterror** command and other than one during parsing of the **iferr** command)
- a result fails to match an **expectvalue** command
- a result matches an **expectbug** command

Whenever any of these conditions arise, any non-empty post-error buffers are executed. Execution starts with buffer 1, and proceeds to buffer 10, executing each non-empty post-error buffer in order.

If an error occurs while executing one of post-error buffers, execution proceeds to the next non-empty post-error buffer. Error and result checking implied by **display resultcheck**, **display errorcheck**, **expectvalue**, etc., are not performed while executing from post-error buffers.

If a post-error buffer contains a command that would terminate the topaz process, then later buffers will have no effect. If a post-error buffer contains a command that would terminate the session, execution later buffers will be attempted but they will not have a session, unless one of the contains "login".

To remove the contents of a specific post-error buffer, enter **iferr** *bufferNumber* without a final argument. The command **iferr\_clear** will clear all buffers.

The **iferr\_list** command will display the contents of all post-error buffers.

The following example uses **expecterror** to test for an error returned by the **printit** command. If Topaz finds one, it displays the active call stack for debugging. That behavior is specified by making the Topaz **stack** command an argument on the **iferr** command line.

```
topaz 1> iferr 1 stack
topaz 1> expecterror GemStoneError 2109
topaz 1> printit
...
%
```

## IFERR\_CLEAR

The **iferr\_clear** command clears all the post-error command buffers.

For details on the post-error command buffers, see the **iferr** command on page 94.

## IFERR\_LIST

The **iferr\_list** command prints all the non-empty post-error command buffers.

For details on the post-error command buffers, see the **iferr** command on page 94.

## IFERROR [*aTopazCommandLine*]

The **iferror** command saves *aTopazCommandLine* to the post-error command buffer 1, or when used without an argument, clearing buffer 1.

The command:

```
topaz 1> iferror stack
```

has the same effect as:

```
topaz 1> iferr 1 stack
```

For details **iferr** and the post-error command buffers, see page 94.

## IMPLEMENTORS *selectorSpec*

Displays a list of all classes that implement the given *selectorSpec* (either a String or a Symbol). This command is equivalent to the following:

```
topaz 1> doit  
ClassOrganizer new implementorsOfReport: aString  
%
```

Before issuing this command, you may need to increase the value of the GEM\_TEMPOBJ\_CACHE\_SIZE configuration parameter beyond its default. For details about GemStone configuration parameters, see the *System Administration Guide for GemStone/S 64 Bit*.

## INPUT [*aFileName* | POP]

Controls the source from which Topaz reads input. Normally Topaz reads input from standard input (stdin). This command causes Topaz to take input from a file or device of your choice.

If you specify a host environment name such as `$HOME/foo.bar` as the input file, Topaz expands that name to the full filename.

If you don't provide an explicit path specification, Topaz looks for the named input file in the directory where you started Topaz.

### **input** *aFileName*

Reads input from the specified file. This pushes the current input file onto a stack and starts Topaz reading from the given file. There is a limit of 20 nested **input** *aFileName* commands. If you exceed the limit, an error is displayed, and execution continues in the current file.

### **input pop**

Pops the current input file from the stack of input files and resumes reading from the previous file. If there is no previous file, or the previous file cannot be reopened, Topaz once again takes its input from standard input.

## INSPECT [*anObjectSpec*]

Sends the message `describe` to the designated object. This command is equivalent to the following

```
topaz 1> send anObjectSpec describe  
%
```

## INTERP

Sends the text following the **interp** command to GemStone for execution as GemStone Smalltalk code, and displays the result.

If there is an error in your code, Topaz displays an error message instead of a legitimate result.

GemStone Smalltalk text is terminated by the first line that contains a % in column 1. For example:

```
topaz 1> interp  
2 + 2  
%  
4
```

The text executed between the **interp** and the terminating % can be any legal GemStone Smalltalk code, and follows the behavior documented in the *Programming Guide for GemStone/S 64 Bit*.

This command is identical to the **run** command (page 139), with one exception: here, execution takes place in interpreted mode.

## **INTERPENV**

This command is used with Ruby applications, but not with Smalltalk applications.

—  
|

## **LEVEL** *anIntegerLevel*

Sets the Topaz display level; that is, this command tells Topaz how much information to include in the result display. A level of 1 (the default) means that the first level of instance variables within a result object will be displayed. Similarly, a level of 2 means that the variables *within* those variables will be displayed. Setting the level to 0 inhibits the display of objects (though object headers will still be displayed if you specify **display oops**). The maximum display level is 32767.

Note the following:

- The **run** command (page 139) displays results using the current display level, as set by the **level** command.
- The **doit** command (page 73) always displays results at level 0, regardless of the current display level.
- The **printit** command (page 131) always displays results at level 1, regardless of the current display level.

## LIMIT [BYTES | OOPS] *anInteger*

Tells Topaz how much of any individual object to display in GemStone Smalltalk results. For example, a limit of 80 would tell Topaz to display no more than 80 bytes (or oops) of any individual object. Setting a limit of 0 tells Topaz not to limit the size of the output. By default, Topaz attempts to display all of an object, no matter how long.

**limit** *anInteger*

**limit bytes** *anInteger*

Tells Topaz how much of any byte object (instance of String or one of String's subclasses) to display in GemStone Smalltalk results.

If *anInteger* is non-zero, then when displaying frame temporaries, or when displaying an object with a display level of 1 or greater, any byte-valued instance variable with a byte object value will be limited to one line (about 80 characters) of output. To display the full contents of that byte object (up to the limit set by *anInteger*), use the **object** command.

For debugging source code, we suggest `limit bytes 5000`.

**limit oops** *anInteger*

Tells Topaz how much of any pointer or nonsequenceable collection to display in GemStone Smalltalk results.

## LIST

The **list** command is used in conjunction with the **set** and **edit** commands to browse through dictionaries, classes, and methods in the repository. The **list** command is also useful in debugging.

When no arguments are included on the command line, the **list** command lists the source code for the currently selected stack frame, as selected by the most recent **up**, **down**, or **frame** command.

### Browsing Dictionaries and Classes

#### **list dictionaries**

Lists the SymbolDictionaries in your GemStone symbol list. This executes the GemStone Smalltalk method `UserProfile>>dictionaryNames`.

#### **list classesIn:** *aDictionary*

Lists the classes in *aDictionary*. For example,

```
topaz 1> list classesIn: UserGlobals
```

lists all of the classes in your UserGlobals dictionary.

#### **list classes**

Lists all of the classes in all of the dictionaries in your symbol list.

#### **list categoriesin:** [*aClass*]

Lists all of the instance and class method selectors for class *aClass*, by category, and establishes *aClass* as the current class for further browsing.

If you omit the class name parameter, method selectors are listed by category for the current class.

#### **list icategories** [*className*]

Lists all of the instance method selectors for the named class, by category. If you specify a class name, that class becomes the current class for subsequent Topaz commands. If you omit the class name parameter, lists the categories of the current class.

#### **list ccategories:** [*className*]

Lists all of the class method selectors for the named class, by category. If you specify a class name, that class becomes the current class for subsequent Topaz commands. If you omit the class name parameter, lists the categories of the current class.

#### **list selectors**

Lists selectors of all instance methods. May be abbreviated as `list sel`.

**list cselectors**

Lists selectors of all class methods. May be abbreviated as `list csel`.

**Listing Methods****list selectorSpec****list method:** *selectorSpec*

Lists the category and source code of the specified instance method selector for the current class. You can also enter this command as `list imethod:`.

For any method whose selector is the same as, or is some subset of, one of the **list** subcommands (for example, a method with the selector **steps**) you must explicitly include the **method:** keyword. For example:

```
topaz 1> list method: steps           (not list steps)
```

**list method:** *@anObjectSpec*

Lists the category and source of the method with the given `objectId`. You can also enter this command as `list imethod:`.

**list classmethod:** *selectorSpec*

Lists the category and source of the given class method selector for the current class. You can also enter this command as `list cmethod:`.

**list classmethod:** *@anObjectSpec*

Lists the category and source of the method with the given `objectId`. You can also enter this command as `list cmethod:`.

**list**

Lists the source code of the active method context. See Chapter 2, “Debugging Your GemStone Smalltalk Code.”

**list @anObjectSpec**

Lists the source code of the `GsNMethod` or `ExecBlock` with the specified `objectId`. That method, or the block’s home method, becomes the default method for subsequent `list` or `disassem` commands.

**Listing Step Points****list steps**

Lists the source code of the active method context, and displays step points in that source code.

**list steps method:** *selectorSpec*

Lists the source code of the specified instance method for the current class, and displays all step points (allowable breakpoints) in that method. For example:

```
topaz 1> set class String
topaz 1> list steps method: includesValue:
  includesValue: aCharacter
* ^1
                                     *****
  "Returns true if the receiver contains aCharacter, false
  otherwise. The search is case-sensitive."
  <primitive: 94>
  aCharacter _validateClass: AbstractCharacter .
*           ^2
                                     *****
  ^ self includesValue: aCharacter asCharacter .
* ^5      ^4
                                     ^3
                                     *****
```

You can use the **break** command to set method breakpoints before assignments, message sends, or method returns. As shown here, the position of each method step point is marked with a caret and a number. Each line of step point information is indicated by asterisks (\*).

For more information about method step points, see Chapter 2, "Debugging Your GemStone Smalltalk Code."

**list steps classmethod:** *selectorSpec*

Lists the source code of the specified class method for the current class, and displays all step points in that method.

## Listing Breakpoints

You can use the **break list** command to list all currently set breakpoints. For more information about using breakpoints, see Chapter 2, "Debugging Your GemStone Smalltalk Code".

**list breaks**

Lists the source code of the active method context, and displays the step points for the method breakpoints currently set in that method. Disabled breakpoints are displayed with negative step point numbers.

**list breaks method:** *selectorSpec*

Lists the source code of the specified instance method for the current class, and displays the method breakpoints currently set in that method. For example:

```
topaz 1> list breaks method: <
< 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'
"
```

```
<primitive: 28>
```

```
aCharCollection _validateClass: CharacterCollection .
*                ^2                *****
^ aCharCollection > self
```

**list breaks classmethod:** *selectorSpec*

Lists the source code of the specified class method for the current class, and displays the method breakpoints currently set in that method.

## LISTW

For the method implied by the current stack frame, limit the list to the number of source lines defined by the **set listwindow** command. The list is centered around the current insertion point for the frame.

If Topaz has a valid session, the **listw** command can be abbreviated as **l**.

## **LOADUA** *aFileName*

Loads the application user action library specified by *aFileName*. This command must be used before **login**. This command cannot be abbreviated.

User action libraries contained user-defined C functions to be called from GemStone Smalltalk. See the *GemBuilder for C* manual for information about dynamically loading user action libraries.

## LOGIN

Lets you log in to a GemStone repository. Before you attempt to log in to GemStone, you'll need to use the **set** command – either interactively or in your Topaz initialization file – to establish certain required login parameters. The required parameters for network communications are:

**set gemnetid:**

name of the GemStone service on the host computer (defaults to `gemnetobject` for the RPC version (**topaz** command) or `gcilnkobj` for the linked version (**topaz -l** command))

**set gemstone:**

name of the Stone (repository monitor) process, including node and protocol information in the form of a network resource string, if necessary. Appendix B describes network resource string syntax.

**set username:**

your GemStone user ID.

**set password:**

your GemStone password. If you do not specify a password (for security reasons, for example), Topaz prompts you for it.

**set hostusername:**

your user account on the host computer. Required for the RPC version of Topaz or for RPC sessions spawned by the linked version.

**set hostpassword:**

your password on the host computer. Required for the RPC version of Topaz or for RPC sessions spawned by the linked version of Topaz. If you enter this command without a password, Topaz prompts you for it.

Topaz allows you to run your Gem (GemStone session), Stone (repository monitor), and Topaz processes on separate network nodes. For more information about this, see the discussion of **set gemnetid** and **set gemstone**.

If you are using linked Topaz (**topaz -l**), also note the following:

- If the `gemnetid` is set to anything other than " (null) or `gcilinkobj`, Topaz starts an RPC session instead of a linked one.
- Topaz can only be linked with a single GemStone session process. If you issue the **login** command to create multiple sessions, the new sessions are RPC rather than linked.

- You cannot use the **set** command to run Gem and Topaz on separate nodes for the linked session. However, you may still run the Stone process on a separate node. For any RPC sessions started from the linked version, you may run the Gems on separate nodes from Topaz.

For more information about logging in to GemStone, see the description of **set** on page 144. Also see the section of Chapter 1 entitled "Logging In to GemStone."

## LOGOUT

Logs out the current GemStone session. This command aborts your current transaction. Your local variables (created with the **define** command) will no longer have valid definitions when you log in again.

Although you can abbreviate most other Topaz commands and parameter names, **logout** must be typed in full.

## LOGOUTIFLOGGEDIN

If logged in, logs out the current GemStone session. If there is no current session, does not increment the Topaz error count.

As with **logout** (page 113), this command aborts your current transaction. Your local variables (created with the **define** command) will no longer have valid definitions when you log in again.

## LOOKUP (METH | METHOD | CMETH | CMETHOD)

*selectorSpec*

**LOOKUP** *className* [**CLASS**] *selectorSpec*

The **lookup** command is used in conjunction with the **set** command to search upwards through the hierarchy of superclasses to locate the implementation of a given method.

### Finding and Listing Methods

**lookup classmethod** *selectorSpec*

Lists the source code of the specified class method for the current class, or searching the superclasses, the first superclass that implements this method. (May be abbreviated as **lookup cmeth**.)

**lookup method** *selectorSpec*

Lists the source code of the specified instance method for the current class, or searching the superclasses, the first superclass that implements this method. (May be abbreviated as **lookup meth**.)

```
topaz 1> set class Symbol
topaz 1> lookup meth match:
```

```
category: 'Comparing'
method: CharacterCollection
match: prefix
```

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

```
self size == 0 ifTrue: [ ^ prefix size == 0 ].
^ self at: 1 equals: prefix
%      [GsMethod objId 2198273]
```

**lookup** *className* *selectorSpec*

Lists the source code of the specified instance method for the given class, or searching the superclasses, the first superclass that implements this method. (The *className* argument may not be **meth**, **method**, **cmeth**, or **classmethod**.)

**lookup** *className* **class** *selectorSpec*

Lists the source code of the specified class method for the given class, or searching the superclasses, the first superclass that implements this method. (The *className* argument may not be **meth**, **method**, **cmeth**, or **classmethod**.)

—  
|

## **METHOD[: *aClassName*]**

Compiles an instance method for the class whose name is given as a parameter. The class of the method you compile will automatically be selected as the current class. If you don't supply a class name, the method is compiled for the current class, as defined by the most recent **set class**;, **list categoriesin**;, **method**;, **classmethod**;, **removeAllMethods**;, **removeAllClassMethods**;, or **fileout class** command.

Text of the method should follow this command on subsequent lines. The method text is terminated by the first line that contains a % character in column 1. For example:

```
topaz 1> method: Animal
habitat
  ^habitat
%
```

Topaz sends the method's text to GemStone for compilation and inclusion in the current category of the specified class. If you haven't yet selected a current category, the new method is inserted in the default category, "as yet unspecified."

## NBRESULT

Wait for and display the result of a previous **nbrun** call. This call may be preceded by a **set session** to switch to the session of an outstanding **nbrun**; otherwise, the current Topaz session is used.

May be immediately preceded by **expectvalue** or **expectbug**, provided that the **expect** commands contain only Integers or numerically coded OOPS (i.e. @NNN), so that no GemStone code is executed before the **nbresult**.

If the **nbrun** has compilation errors, those will be displayed by the **nbresult**. If there is no outstanding **nbrun** for the session the result is:

```
[276 sz:0 cls: 76289 UndefinedObject] remoteNil
```

Note that nonblocking operations do block in linked sessions, and in a linked session the result with no outstanding **nbrun** is `nil`, not `remoteNil`.

This command is the equivalent of calling the GemBuilder for C function **GciNbEnd**.

## NBRUN

Similar to **run**, but execution is nonblocking, so the application can proceed with non-GemStone tasks while the expression is executed. To get the results of the execution, use **nbresult**.

In a linked session, **nbrun** is blocking (necessarily). In this case a warning message is displayed. For example:

```
topaz 1> nbrun
Time now
%
Current session not remote, nbrun executing synchronously
topaz 1> nbresult
09:48:17
```

**nbrun** should not be immediately preceded by **expect** commands, since this command has no result. May be followed by a **set session** and another **nbrun** to start an execution in another session.

The text of this command is not accessible from **edit last**.

This command is the equivalent of calling the GemBuilder for C function **GciNbExecute**.

## NBSTEP

Similar to **step**, but execution is nonblocking. To get the results of the execution, use **nbresult**.

In a linked session, **nbstep** is blocking (necessarily). In this case a warning message is displayed.

Should not be immediately preceded by **expect** commands, since this command has no result. May be followed by a **set session** and another **nbrun** or **nbstep** to start an execution in another session.

This command is the equivalent of calling the GemBuilder for C function **GciNbStep**.

## **OBJ1** *anObjectSpec*

## **OBJ2** *anObjectSpec*

Equivalent to the **object** command, but with the following difference: results are displayed at level 1 (if `obj1`) or level 2 (if `obj2`), with offsets of instance variables shown as one-based. After execution, previous settings for `level` and `omit|display zerobased` are restored.

**OBJ1Z** *anObjectSpec*

**OBJ2Z** *anObjectSpec*

Equivalent to the **object** command, but with the following difference: results are displayed at level 1 (if `obj1`) or level 2 (if `obj2`), with offsets of instance variables shown as zero-based. After execution, previous settings for `level` and `omit|display zerobased` are restored.

—  
|

**OBJECT** *anObjectSpec* [**AT:** *anIndex* [**PUT:** *anObjectSpec*]]

Provides structural access to GemStone objects, allowing you to peek and poke at objects without sending messages. The first *anObjectSpec* argument is an object specification in one of the Topaz object specification formats. All formats described in “Specifying Objects” on page 42 are legal in **object** commands.

You can use local variables (created with the **define** command) in **object** commands. The local definition of a symbol always overrides any definition of the symbol in GemStone. For example, if you defined the local variable `thirdvar`, and your UserGlobals dictionary also defined a GemStone symbol named `thirdvar`, the definition of that GemStone symbol would be ignored in **object** commands.

**object** *anObjectSpec at:anIndex*

Returns the value of an instance variable within the designated object at the specified integer offset. You can string together **at:** parameters after **object** to descend as far as you like into the object of interest.

As far as **object at:** is concerned, named and indexed instance variables are both numbered, and indexed instance variables follow named instance variables when an object has both. That is, if an indexable object also had three named instance variables, the first indexed field would be addressed with `object theIdxObj at:4`.

Nonsequenceable collections are also considered indexable via **object at:**.

**object** *anObjectSpec at:anIndex put:anotherObjectSpec*

Lets you store values into instance variables. This command stores the second *anObjectSpec* object into the first *anObjectSpec* object at the specified integer offset.

You cannot store into an NSC with **object at: put:**, although you can scrutinize its elements with **object at:**.

**CAUTION**

*Because **object at: put:** bypasses all the protections built into the GemStone Smalltalk kernel class protocol, you risk corrupting your repository whenever you permanently modify objects with this command.*

The following example shows how you could use **object at: put:** to store a new String in MyAnimal's *habitat* instance variable:

```
topaz 1> object MyAnimal at: 3 put: 'pond'
an Animal
  name           nil
  favoriteFood   nil
  habitat        pond
```

Like **object at:**, the **object at: put:** command can take a long sequence of parameters. For example:

```
topaz 1> object MyAnimal at: 3 at: 1 put: $1
liver
```

This example stores the character "l" into the first instance variable of MyAnimal's third instance variable.

With this command you can store Characters or SmallIntegers in the range from 0–255 (inclusive) into a byte object. You can also store other byte objects such as Strings. For example:

```
topaz 1> object 'this' at: 5 put: ' and that'
this and that
```

The **object at: put:** command behaves differently for objects with byte-array and pointer-array implementations. You may store the following kinds of objects into byte-array type objects:

**Character.** This stores the character '9':

```
topaz 1> object '123' at: 1 put: $9
```

**SmallInteger.** This stores a byte with the value 48:

```
topaz 1> object '123' at: 1 put: 48
```

**Byte arrays.** This stores 'b' and 'c' at offsets 2 and 3:

```
topaz 1> object '1234' at: 2 put: 'bc'
```

## OMIT *aDisplayFeature*

The **display** and **omit** commands control the display of instance variable names, hexadecimal byte values, and OOPs (object-oriented pointers). The **omit** command turns off these display attributes, and the **display** command turns them on.

### **omit oops**

Do not display OOP values with displayed results. (This is the default condition.)

### **omit bytes**

When displaying string objects, do not include the hexadecimal value of each byte. (This is the default condition.)

### **omit errorcheck**

Disables automatic result recording, stopping the effect of **display errorcheck**. Closes the `./topazerrors.log` file.

### **omit names**

For each of an object's named instance variables, do not display the instance variable's name along with its value. When you have issued **omit names**, named instance variables appear as `i1`, `i2`, `i3`, etc.

### **omit resultCheck**

Disables automatic result checking, stopping the effect of **display resultCheck**. Closes the `./topazerrors.log` file and stops checking the results of successful **run**, **printit**, etc. commands. You can still check the result of an individual **run** command by entering an **expectvalue** command just before it.

### **omit pauseonerror**

Disables pauses in Topaz execution after errors, stopping the effect of **display pauseonerror**. When pause-on-error mode is turned off, the **status** command output includes:

```
omit interactive pause on errors
```

### **omit classoops**

Disables the display of OOPs of classes along with class names in object display, stopping the effect of **display classoops**.

### **omit lineeditor**

Disables the use of the Topaz line editor, stopping the effect of **display lineeditor**. (Not available on Windows.)

**omit pushonly**

Disables the effect of the **only** keyword in an **object push** command, stopping the effect of **display pushonly**.

**omit zerobased**

Shows offsets of instance variables as one-based when displaying objects. (This is the default condition.) To show offsets as zero-based, use the **display zerobased** command.

—  
|

## OUTPUT (PUSH | APPEND | PUSHNEW | POP)

### *aFileName* [ONLY]

Controls where Topaz output is sent. Normally Topaz sends output to standard output (stdout). This command redirects all Topaz output to a file (or device) of your choice.

If you specify a host environment name such as `$HOME/foo.bar` as the output file, Topaz expands that name to the full filename. If you don't provide an explicit path specification, Topaz output is sent to the named file in the directory where you started Topaz.

As the command names **push** and **pop** imply, Topaz can maintain a stack of up to 20 output files, with current interactions captured in the file on top of the stack.

**output** *aFileName*

**output push** *aFileName*

Sends output to the specified file. If the file you name doesn't yet exist, Topaz will create it. If you name an existing file, Topaz overwrites it.

To append output to an existing file, precede the file name with an ampersand (&).

Although you can abbreviate most other Topaz commands and parameter names, **push** must be typed in full.

**output append** *aFileName*

Sends output to the specified file. If the file you name doesn't yet exist, Topaz will create it. If you name an existing file, Topaz will append to it. This behavior is the same as **output push &aFileName**.

Although you can abbreviate most other Topaz commands and parameter names, **append** must be typed in full.

**output pushnew** *aFileName*

Sends output to the specified file. If the file you name doesn't exist, Topaz will create it. If you name an existing file, Topaz will create a new file. For a filename of the form `foo.out`, the new filename will be `foo_N.out`, where `N` is some integer between 1 and 100 (inclusive), and where `foo_N.out` did not previously exist. If more than 100 versions of the file exist, the oldest version will be overwritten.

Although you can abbreviate most other Topaz commands and parameter names, **pushnew** must be typed in full.

**output** *aFileName* **only**

**output push** *aFileName* **only**

**output append** *aFileName* **only**

**output pushnew** *aFileName* **only**

Sends output to the specified file, but does not echo that output to standard output (usually, your screen).

**output pop**

Stops output to the current output file (that is, the file most recently named in an **output push** command). The file is closed, and output is again sent to the previously named output file. If there is no previous output file, an error message is issued and the I/O stacks are reset.

Although you can abbreviate most other Topaz commands and parameter names, **pop** must be typed in full.

## **PAUSEFORDEBUG** [*errorNumber*]

Provided to assist internal debugging of a session.

With no argument, this command has no effect.

## **PKGLOOKUP (METH | METHOD | CMETH | CMETHOD)** *selectorSpec*

**PKGLOOKUP** *className* [**CLASS**] *selectorSpec*

Similar to the **lookup** command, but with one key exception: **pkglookup** looks first in GsPackagePolicy state, then in the persistent method dictionaries for each class up the hierarchy. The **pkglookup** command does not look at transient (session method) dictionaries.

For details, see the description of the **lookup** command on page 115.

## PRINTIT

Sends the text following the **printit** command to GemStone for execution as GemStone Smalltalk code, and displays the result. If there is an error in your code, Topaz displays an error message instead of a legitimate result. GemStone Smalltalk text is terminated by the first line that contains a % in column 1. For example:

```
topaz 1> printit
2 + 2
%
4
```

The text executed between the **printit** and the terminating % can be any legal GemStone Smalltalk code, and follows all the behavior documented in the *GemStone/S Programming Guide*.

If the configuration parameter GEM\_NATIVE\_CODE\_ENABLED is set to FALSE, or if any breakpoints are set, execution defaults to interpreted mode. Otherwise, execution defaults to using native mode.

- For details about GemStone configuration parameters, see the *System Administration Guide for GemStone/S 64 Bit*.

Note that **printit** always displays results at level 1, regardless of the current display level setting (page 103). The **printit** command does not alter the current level setting. The **run** command (page 139) displays according to the current level setting, and the **doit** command (page 73) displays results at level 0.

## PROTECTMETHODS

After this command, all subsequent method compilations during the current session must contain either a <protected> or <unprotected> directive.

Used for consistency checking in filein scripts.

## QUIT [*aSmallInt* / *anObjectSpec*]

Leaves Topaz, returning to the operating system. If you are still logged in to GemStone when you type **quit**, this aborts your transaction and logs out all active sessions. Although you can abbreviate most other Topaz commands and parameter names, **quit** must be typed in full.

You can include an argument (a `SmallInteger`, or an object specification that resolves to a `SmallInteger`) to specify an explicit `exitStatus` for the Topaz process. If you do not specify this argument, the `exitStatus` will be either 0 (no errors occurred during Topaz execution) or 1 (there was a GCI error or the Topaz `errorCount` was nonzero).

## RELEASEALL

Empty Topaz's internal buffer of object identifiers (the export set). Objects are placed in the export set as a result of object creation and certain other object operations. **releaseall** is performed automatically prior to each **run**, **doit**, **printit**, or **send**.

For more information, see the *GemStone/S 64 Bit GemBuilder for C* manual. This is equivalent to the GemBuilder for C call **GciReleaseOops**.

## **REMARK** *commentText*

Begins a remark (comment) line. Topaz ignores all succeeding characters on the line. You can also use an exclamation point (!) in column 1 of a line to signal the beginning of a comment. Comments are often useful in annotating Topaz batch processing files, such as test scripts.

## REMOVEALLCLASSMETHODS [*aClassName*]

Removes all class methods from the class whose name you give as a parameter. The specified class automatically becomes the current class.

If you don't supply a class name, the methods are removed from the current class, as defined by the most recent **set class;**, **list categoriesin;**, **method;**, or **classmethod:** command.

Although you can abbreviate most other Topaz commands and parameter names, **removeAllClassMethods** must be typed in full.

## REMOVEALLMETHODS [*aClassName*]

Removes all instance methods from the class whose name you give as a parameter. The specified class automatically becomes the current class.

If you don't supply a class name, the methods are removed from the current class, as defined by the most recent **set class**;, **list categoriesin**;, **method**;, or **fileout class** command.

Although you can abbreviate most other Topaz commands and parameter names, **removeAllMethods** must be typed in full.

**RUBYCLASSMETHOD**  
**RUBYHIERARCHY**  
**RUBYIMPLEMENTORS**  
**RUBYLIST**  
**RUBYLOOKUP**  
**RUBYMETHOD**  
**RUBYRUN**

These commands are used with Ruby applications, but not with Smalltalk applications.

## RUN

Sends the text following the **run** command to GemStone for execution as GemStone Smalltalk code, and displays the result.

If there is an error in your code, Topaz displays an error message instead of a legitimate result.

GemStone Smalltalk text is terminated by the first line that contains a % in column 1. For example:

```
topaz 1> run
2 + 2
%
4
```

The text executed between the **run** and the terminating % can be any legal GemStone Smalltalk code, and follows all the behavior documented in the *GemStone/S Programming Guide*.

If the configuration parameter GEM\_NATIVE\_CODE\_ENABLED is set to FALSE, or if any breakpoints are set, execution defaults to interpreted mode. Otherwise, execution defaults to using native mode. For details about GemStone configuration parameters, see the *System Administration Guide for GemStone/S 64 Bit*.

The **run** command is similar to **printit**, with one significant difference. The **run** command uses the current display level setting (page 103), whereas **printit** always displays the result as if level 1 were the most recent **level** command.

## RUNENV

This command is used with Ruby applications, but not with Smalltalk applications.

—  
|

## SEND *anObjectSpec aMessage*

Sends a message to an object.

The **send** command's first argument is an object specification identifying a receiver. The object specification is followed by a message expression built almost as it would be in GemStone Smalltalk, by mixing the keywords and arguments. For example:

```
topaz 1> level 0
topaz 1> send System myUserProfile
a UserProfile
topaz 1> send 1 + 2
3
topaz 1> send @10443 deleteEntry: @33234
```

There are some differences between **send** syntax and GemStone Smalltalk expression syntax. Only one message send can be performed at a time with **send**. Cascaded messages and parenthetical messages are not recognized by this command. Also, each item must be delimited by one or more spaces or tabs.

All Topaz object specification formats (as described in "Specifying Objects" on page 42) are legal in **send** commands.

If the configuration parameter `GEM_NATIVE_CODE_ENABLED` is set to `FALSE`, or if any breakpoints are set, execution defaults to interpreted mode. Otherwise, execution defaults to using native mode.

- For details about GemStone configuration parameters, see the *System Administration Guide for GemStone/S 64 Bit*.

## SENDENV

This command is used with Ruby applications, but not with Smalltalk applications.

—  
|

## **SENDERS** *selectorSpec*

Displays a list of all classes that are senders of the given *selectorSpec* (either a String or a Symbol). This command is equivalent to the following

```
topaz 1> doit  
ClassOrganizer new sendersOfReport: aString  
%
```

Before issuing this command, you may need to increase the value of the GEM\_TEMPOBJ\_CACHE\_SIZE configuration parameter beyond its default. For details about GemStone configuration parameters, see the *System Administration Guide for GemStone/S 64 Bit*.

## SET *aTopazParameter* [*aParamValue*]

The **set** command allows you to select a class and category to work with in examining, modifying, and creating classes with the **list** and **edit** commands. You'll also use **set** in establishing your GemStone login parameters.

You can combine two or more set items on one command line, and you can abbreviate token names to uniqueness. For example:

```
topaz 1> set gemstone gs64stone user DataCurator
```

### **set cachename:** *aString*

This option is valid for linked sessions only, not for RPC sessions. The colon is not required.

Sets the name that will be used for this session in cache statistics collected by statmonitor. Setting the name prior to login allows statistics to be collected and displayed under a single meaningful name, rather than being split between the initial default name and a later meaningful name assigned using `System class >> cacheName::`.

### **set category:** *aCategory*

Sets the current category, the category for subsequent method compilations. You must be logged in to use this command. If you try to compile a method without first selecting a category, the new method is inserted in the default category "as yet unspecified." The **set category:** command has the same effect as the **category:** command.

If the category you name doesn't already exist, Topaz will create it when you first compile a method.

Specifying a new class with **set class** does not change your category. However, when you **edit** or **fileout** a method, that method's category becomes the current category.

The current category is cleared by the **logout**, **login**, and **set session** commands.

### **set class:** *aClassName*

Sets the current class. You must be logged in to use this command. After setting the current class, you can list its categories and methods with the **list categories** command. You can select a category to work with through either the **set category:** or **category:** command.

The current class may also be redefined by the **list categoriesin**, **method**, **classmethod**, **removeAllMethods**, **removeAllClassMethods**, and **fileout class**: commands.

The current class is cleared by the **logout**, **login**, and **set session** commands.

To display the name of the current class, issue the **set class** command without a class name.

**set compile\_env**: *anInteger*

Not normally used in Smalltalk. Sets the compilation environmentId used for method complications and **run**, **printit**, etc. *anInteger* must be between 0 and 255 and is 0 by default.

**set editorname**: *aHostEditorName*

Sets the name of the editor you want to use in conjunction with the **edit** command. For example:

```
topaz 1> set editorname: vi
```

The default is set from your \$EDITOR environment variable, if it is defined.

**set gemnetid**: *aServiceName*

*aServiceName* is a network resource string specifying the name of the GemStone service (that is, the host process to which your Topaz session will be connected) and its host computer.

For the RPC version of Topaz the default **gemnetid** parameter is `gemnetobject`, which is the GemStone service name in most GemStone installations. You may also use `gemnetdebug` or your own custom gem service. RPC versions of Topaz cannot start linked sessions.

For linked Topaz (started with **topaz -l**), the default **gemnetid** is `gcilnkobj`. Use the `status` command to verify that this parameter is `gcilnkobj` or `"`. This makes the first session to log in a linked session. It is only possible to have one linked session per topaz process. If **gemnetid** is set to a gem service such as `gemnetobject`, **topaz -l** starts RPC sessions. In this case, the lowest number for the prompt is `topaz 2>`, because `topaz 1>` is reserved for a linked session. After you start the RPC session you can still start a linked session by resetting the **gemnetid** to an empty string:

```
set gemnetid: ''
```

You can run your GemStone session (Gem), repository monitor (Stone) process, and your Topaz processes on separate nodes in your network. The one exception is the linked Topaz session, when Topaz and the Gem run as a single process. Network resource strings allow you to designate the nodes on

which the Gem and Stone processes run. For example, a Gem process called `gemnetobject` on node `lichen` could be described in network resource string syntax as:

```
!@lichen!gemnetobject
```

To specify a Gem running on the current node, omit the *node* portion of the string, and specify only the Gem name: `gemnetobject`. Appendix B describes network resource string syntax.

**set gemstone:** *aGemStoneName*

Specifies the name of the GemStone you want to log in to. The standard name is `gs64stone`; if this doesn't work for you, see your GemStone data curator.

You can run your GemStone session (Gem), repository monitor (Stone) process, and your Topaz processes on separate nodes in your network. The one exception is the linked Topaz session, when Topaz and the Gem run as a single process. Network resource strings allow you to designate the nodes on which the Gem and Stone processes run. For example, a Stone process called `gs64stone` on node `lichen` could be described in network resource string syntax as:

```
!@lichen!gs64stone
```

To specify a Stone running on the same node as the Gem, omit the *node* portion of the string, and specify only the Stone name: `gs64stone`. Appendix B describes network resource string syntax.

**set history:** *anInt*

Sets the history size of the Topaz line editor. The argument *anInt* may be between 0 and 1000, inclusive. (Not available on Windows.)

**set hostpassword:** *aPassword*

Sets the host password to be used when you next log in. If you don't include the password on the command line, Topaz prompts you for it. Prompted input taken from the terminal is not echoed. This lets you put a **set hostpassword:** command in your Topaz initialization file so that Topaz automatically prompts you for your password. Note, however, that this command must *follow* the **set hostusername:** command.

With TCP/IP, if you do not explicitly supply a host username and password, Topaz will try to find a username for the designated node in a file in your home directory. See the following discussion of **set hostusername**.

For a linked Topaz session, **set hostpassword** has no effect, because no separate Gem process is created on the host computer. The password is

required, however, if you spawn new sessions while you are running linked Topaz, because the additional sessions are always RPC Topaz.

**set hostusername:** *aUsername*

Sets the account name you use when you log in to the host computer. When you run Topaz, a Gem (GemStone session) process is started on the host computer specified by the **set gemnetid:** command. The **set hostusername:** command tells Topaz which account you want that process to run under.

If you do not explicitly supply a host username and password, Topaz tries to find a username and password for the designated node in a network initialization file in your home directory. Under UNIX, that file is `$HOME/.netrc` and should contain lines of the form

```
machine aNode login aUsername password aPassword
```

For example, `$HOME/.netrc` under UNIX:

```
machine alf login joebob password mypassword
```

Because the network initialization file contains your password, you should ensure that others — group or world — do not have authorization to read it.

To clear the hostusername field, enter:

```
topaz 1> set hostusername *
```

For a linked Topaz session, **set hostusername** has no effect, since no separate Gem process is created on the host computer.)

**set listwindow:** *anInt*

Defines the maximum number of source lines to be listed by the **listw** command (page 109).

**set nrsdefaults:** *aNRShheader*

Sets the default components to be used in network resource string specifications. The parameter *aNRShheader* is a network resource string header that may specify any NRS modifiers' default values. The initial value of **nrsdefaults** is the value of the `GEMSTONE_NRS_ALL` environment variable. The Topaz **status** command shows the value of **nrsdefaults** unless it is the empty string.

**set password:** *aGemStonePassword*

Sets the GemStone password to be used when you next log in. If you don't include the password on the command line, Topaz prompts you for it. Prompted input is taken from the terminal and not echoed. This lets you put a **set password:** command in your Topaz initialization file so that Topaz will

automatically prompt you for your password. Note, however, that this command must *follow* the **set username:** command.

**set session:** *aSessionNumber*

Connects Topaz to the session whose ID is *aSessionNumber*. When you log in to GemStone, Topaz displays the session ID number for that connection. This command allows you to switch among multiple sessions. (The Topaz prompt always shows the number of the current session.)

If you specify an invalid session number, an error message is displayed, and the current session is retained.

This command clears the current class and category. After you switch sessions with **set session**, your local variables (created with the **define** command) no longer have valid definitions.

**set sourcestringclass:** *aClass*

Sets the class used to instantiate Smalltalk source strings generated by the **run**, **edit**, **method**, and **classmethod** commands. For example:

```
set sourcestringclass String
set sourcestringclass DoubleByteString
set sourcestringclass IsoLatin
```

The Topaz **status** command shows the current source class. The default source class is String. The **set session** command resets the source string class to the default.

**set stackpad:** *anInt*

Defines the minimum size used when formatting lines in a stack display. The argument *anInt* may be between 0 and 256, inclusive. (Default: 45.)

**set tab:** *anInt*

Defines the number of spaces to insert when translating a tab (CTRL-I) character when printing method source strings. The argument *anInt* may be between 1 and 16, inclusive. (Default: 8.)

**set username:** *aGemStoneUsername*

Establishes a GemStone user ID for the next login attempt. Your GemStone data curator can tell you your user name.

## **SHELL** [*aHostCommand*]

When issued with no parameters, this command creates a child process in the host operating system, leaving you at the operating system prompt. To get back into Topaz, exit the command shell by typing **Control-D** (from the UNIX Bourne or Korn shells), typing **logout** (from the UNIX C shell), or typing **exit** (from a DOS shell).

If you supply parameters on the **shell** command line, they pass to a subprocess as a command for execution. For example:

```
topaz 1> shell ls -l /user1/janec.topaz
total 4
-rw-r--r-- 1 janec 196 Jul 1 22:31 animal.gs
-rw-r--r-- 1 janec 139 Jul 1 22:31 animaltest.log
-rw-r--r-- 1 janec 287 Jul 1 22:31 mathtest.log
-rw-r--r-- 1 janec 110 Jul 1 22:32 mathtest2.log
topaz 1>
```

On UNIX systems, a **shell** command issued without parameters creates a shell of whatever type is customary for the user account (C, Bourne, or Korn). When issued with parameters, **shell** always creates a shell of the system default type (either Bourne or Korn).

## **SPAWN** [*aHostCommand*]

Included for compatibility with previous versions. See the **shell** command on page 149.

—  
|

## STACK [*aSubCommand*]

Topaz can maintain up to eight simultaneous GemStone Smalltalk call stacks that provide information about the GemStone state of execution. Each call stack consists of a linked list of contexts.

The call stack becomes active, and the **stack** command becomes accessible, when you execute GemStone Smalltalk code containing a breakpoint. The **stack** command allows you to display the contexts in the active call stack; redefine the active context for debugging commands within the active call stack; reset whether the active call stack is automatically saved before certain Topaz execution commands are performed; display the top context of each saved call stacks; redefine the active call stack; or remove one or all call stacks.

### Display the Active Call Stack

#### **stack**

Displays all of the contexts in the active call stack, starting with the active context. For each context in the stack display, the following items are displayed:

- the level number
- the class of the GsMethod
- selector of the method
- the environmentId (not used by Smalltalk)
- the current step point (that is, assignment, message send, or method return) within the method
- the line number of the current step point within the source code of the method
- the receiver and parameters for this context.
- the method temporaries (if **display oops** is active)
- the OOP of the GsNMethod (if **display oops** is active)

If any context in the display is a block, the display for that context begins with SimpleBlock or ComplexBlock, as shown in the examples below.

The resulting display is governed by the setting of other Topaz commands such as **limit**, **level**, and **display** or **omit**.

The active context resets to 1 whenever any of the following commands is executed: **run**, **printit**, **send**, **doit**, **step**, **edit last**, or **edit new text**.

Here is an example of the stack display when **display oops** is active:

```

topaz 1> run
{ 1 . 2 } do: [:x | x / 0 ]
%
ERROR 2026 , a ZeroDivide occurred (error 2026),
reason:numErrIntDivisionByZero, An attempt was made to divide nil by
zero. (ZeroDivide)
topaz 1> display oops
topaz 1> stack
==> 1 AbstractException >> _signalWith:          (envId 0) @5 line 25
[methId 4488705]
    receiver [26722305 sz:11 cls: 131329 ZeroDivide] a ZeroDivide
occurred (error 2026), reason:numErrIntDivision...
    handleInCextensionBool [20 sz:0 cls: 76289 UndefinedObject] nil
    res [20 sz:0 cls: 76289 UndefinedObject] nil
    .t1 [26722305 sz:11 cls: 131329 ZeroDivide] a ZeroDivide occurred
(error 2026), reason:numErrIntDivision...
2 AbstractException >> signal                      (envId 0) @2 line 47
[methId 4485889]
    receiver [26722305 sz:11 cls: 131329 ZeroDivide] a ZeroDivide
occurred (error 2026), reason:numErrIntDivision...
3 Number >> _errorDivideByZero                    (envId 0) @6 line 7
[methId 3428865]
    receiver [10 sz:0 cls: 74241 SmallInteger] 1 == 0x1
4 SmallInteger >> /                               (envId 0) @6 line 7
[methId 3574785]
    receiver [10 sz:0 cls: 74241 SmallInteger] 1 == 0x1
    aNumber [2 sz:0 cls: 74241 SmallInteger] 0 == 0x0
5 [] in Executed Code                             (envId 0) @2 line 1
[methId 26714369]
    self [20 sz:0 cls: 76289 UndefinedObject] nil
    receiver [26713089 sz:5 cls: 128001 ExecBlock1] anExecBlock1
    x [10 sz:0 cls: 74241 SmallInteger] 1 == 0x1
6 Collection >> do:                               (envId 0) @5 line 10
[methId 2499585]
    receiver [26710785 sz:2 cls: 66817 Array] anArray
    aBlock [26713089 sz:5 cls: 128001 ExecBlock1] anExecBlock1
    i [10 sz:0 cls: 74241 SmallInteger] 1 == 0x1
    .t1 [18 sz:0 cls: 74241 SmallInteger] 2 == 0x2
    .t2 [10 sz:0 cls: 74241 SmallInteger] 1 == 0x1

```

```

.t3 [26713089 sz:5 cls: 128001 ExecBlock1] anExecBlock1
.t4 [10 sz:0 cls: 74241 SmallInteger] 1 == 0x1
7 Executed Code @2 line 1
[methId 26713601]
receiver [20 sz:0 cls: 76289 UndefinedObject] nil
8 GsNMethod class >> _gsReturnToC (envId 0) @1 line 1
[methId 4464641]
receiver [20 sz:0 cls: 76289 UndefinedObject] nil

```

**stack anInt**

Displays contexts in the active call stack, starting with the active context. The argument *anInt* indicates how much of the stack to display. For example, if *anInt* is 1, this command shows only the active context. If *anInt* is 2, this command also shows the caller of the active context, etc.

## Display or Redefine the Active Context

**stack scope**

Displays the current active context. For example:

```

topaz 1> stack scope
1 AbstractException >> _signalWith: (envId 0) @6 line 25

```

**stack scope anInt**

Redefines the active context within the active call stack and displays the new context. The integer 1 represents the currently active context, while the integer 2 represents the *caller* of the active context.

**stack up**

Moves the current context up one level toward the top of the stack and displays the new context. This is equivalent to the command line:

```
stack scope <current scope + 1>
```

**stack down**

Moves the current scope down one level away from the top of the stack and displays the new context. This is equivalent to the command line:

```
stack scope <current scope - 1>
```

**stack trim**

Trims the stack so that the current scope becomes the new top of the stack. Execution resumes at the first instruction in the method at the new top of the stack. If that method has been recompiled, **stack trim** installs the new version of the method. The new top of the stack must not represent the activation of an ExecutableBlock.

For more about this, see the method comments for  
GsProcess>>\_trimStackToLevel: and GsProcess>>\_localTrimStackToLevel:.

If the stack is trimmed, any resumption of execution will take place in interpreted mode.

## Save or Delete the Active Call Stack During Execution

### **stack nosave**

Causes your active call stack to be deleted before executing any of the following commands: **run**, **printit**, **send**, **doit**, **edit last**, or **edit new text**.

This is the default condition.

### **stack save**

Automatically saves the active call stack before executing any of the commands listed for **stack nosave** (above).

## Display All Call Stacks

### **stack all**

Displays your list of saved call stacks. The list includes the top context of every call stack (stack 1). For example:

```
topaz 1> stack all
*1 Behavior >> new @ 1
 2 Animal >> habitat @ 3
 3 Executed Code @ 2
```

The \* indicates the active call stack, if one exists. If there are no saved stacks, a message to that effect is displayed.

## Redefine the Active Call Stack

### **stack change** *anInt*

Sets the active call stack to the call stack indicated by *anInt* in the **stack all** command output, and displays one frame of the newly selected call stack.

## Remove Call Stacks

**stack delete** *aStackInt*

Removes the call stack indicated by *aStackInt* in the **stack all** command output.

Topaz maintains up to eight simultaneous call stacks. If all eight call stacks are in use, you must use this command to delete a call stack before issuing any of the following commands: **run**, **printit**, **send**, **doit**, **edit last**, or **edit new text**.

**stack delete all**

Removes all call stacks.

## STATUS

Displays your current login settings and other information about your Topaz session.

For example:

```
topaz 1> status

display level: 0
byte limit: 0
omit bytes
display instance variable names
display oops
oop limit: 0
omit automatic result checks
omit interactive pause on errors
listwindow: 20
stackpad: 45
tab (ctl-H) equals 8 spaces when listing method source
using line editor
  line editor history: 100
  topaz input is from a tty on stdin
EditorName_____ vi

CompilationEnv___ 0
Connection Information:
UserName_____ 'Isaac Newton'
Password _____ (set)
HostUserName_____ 'newtoni'
HostPassword_____ (set)
NRSdefaults_____ '#netldi:nldi30'
GemStone_____ 'gs64stone'
GemNetId_____ 'gemnetobject'
GemStone NRS_____ '!#netldi:nldi30#server!gs64stone'
browsing information:
Class_____
Category_____ (as yet unclassified)
Source String Class__ String
```

These settings are set to default values when Topaz starts, and may be modified using the **set** command. See page 144 for details on **set** and the specific settings.

## STEP (OVER | INTO)

Advances execution to the next step point (assignment, message send, or method return) and halts. You can use the step command to continue execution of your GemStone Smalltalk code after an error or breakpoint has been encountered. For examples and other useful information, see Chapter 2, “Debugging Your GemStone Smalltalk Code.”

### **step**

Equivalent to **step over**.

### **step over**

Advances execution to the next step point in the active context or its caller. The active context is the top of the stack or the context specified by the last **frame**, **up**, **down**, **stack scope**, **stack up**, or **stack down** command.

### **step into**

Advances execution to the next step point in your GemStone Smalltalk code.

## STK [*aSubCommand*]

Similar to **stack**, but does not display parameters and temporaries for each context. All contexts for the active call stack are displayed, with the current active context indicated by an arrow.

For more information on *s*, see the **stack** command on page 151.

```
topaz 1> run
{ 1 . 2} do: [:x | x / 0 ]
%
ERROR 2026 , a ZeroDivide occurred (error 2026),
reason:numErrIntDivisionByZero, An attempt was made to
divide nil by zero. (ZeroDivide)
topaz 1> stk
==> 1 AbstractException >> _signalWith: (envId 0) @5 line 25
2 AbstractException >> signal (envId 0) @2 line 47
3 Number >> _errorDivideByZero (envId 0) @6 line 7
4 SmallInteger >> / (envId 0) @6 line 7
5 [] in Executed Code (envId 0) @2 line 1
6 Collection >> do: (envId 0) @5 line 10
7 Executed Code @2 line 1
8 GsNMethod class >> _gsReturnToC (envId 0) @1 line 1
topaz 1> up 3
4 SmallInteger >> / (envId 0) @6 line 7
receiver 1
aNumber 0
topaz 1> stk
1 AbstractException >> _signalWith: (envId 0) @5 line 25
2 AbstractException >> signal (envId 0) @2 line 47
3 Number >> _errorDivideByZero (envId 0) @6 line 7
==> 4 SmallInteger >> / (envId 0) @6 line 7
5 [] in Executed Code (envId 0) @2 line 1
6 Collection >> do: (envId 0) @5 line 10
7 Executed Code @2 line 1
8 GsNMethod class >> _gsReturnToC (envId 0) @1 line 1
```

## **STRINGS** *selectorSpec*

Displays a list of all methods that contain the given *selectorSpec* (either a String or a Symbol) in their source string. This command cannot be abbreviated.

The **strings** command is equivalent to the following:

```
topaz 1> doit  
ClassOrganizer new strings: aString  
%
```

Before issuing this command, you may need to increase the value of the GEM\_TEMPOBJ\_CACHE\_SIZE configuration parameter beyond its default. For details about GemStone configuration parameters, see the *System Administration Guide for GemStone/S 64 Bit*.

## STRINGSIc *selectorSpec*

Displays a list of all methods that contain the given *selectorSpec* (either a String or a Symbol) in their source string. Similar to the **strings** command, with one significant difference: **stringsic** is case-insensitive. This command cannot be abbreviated.

The **stringsic** command is equivalent to the following:

```
topaz 1> doit  
ClassOrganizer new stringsIc: aString  
%
```

Before issuing this command, you may need to increase the value of the GEM\_TEMPOBJ\_CACHE\_SIZE configuration parameter beyond its default. For details about GemStone configuration parameters, see the *System Administration Guide for GemStone/S 64 Bit*.

## **SUBCLASSES** [*aClassName*]

Prints subclasses of the specified class. If you don't specify a class name, prints subclasses of the current class.

## TEMPORARY [*aTempName*[/*anInt*] [*anObjectSpec*] ]

Displays or redefines the value of one or more temporary variables in the currently active context. For examples and other useful information, see Chapter 2, “Debugging Your GemStone Smalltalk Code.”

All Topaz object specification formats (as described in “Specifying Objects” on page 42) are legal in **temporary** commands.

### **temporary**

Displays the names and values of all temporary objects in the active context.

### **temporary** *aTempName*

Displays the value of the first temporary object with the specified name in the currently active context.

```
topaz 1> temporary preferences
preferences      an Array
```

### **temporary** *aTempName anObjectSpec*

Redefines the specified temporary in the currently active context to have the value *anObjectSpec*.

### **temporary** *anInt*

Displays the value of the temporary at offset *n* in the currently active context. Use this form of the command to access a temporary with a duplicate name, because **temporary** *aTempName* always displays the first temporary with the specified name.

### **temporary** *anInt anObjectSpec*

Redefines the temporary at offset *n* in the currently active context to have the value *anObjectSpec*.

For example, to view the temporary variable values:

```
topaz 1> run
String withAll: 'abc'
%
a Breakpoint occurred (error 6005), Method breakpoint
encountered.
1 String class >> withAll:                (envId 0) @1 line 1
topaz 1> stack
==> 1 String class >> withAll:            (envId 0) @1 line 1
    receiver String
    aString abc
2 Executed Code                            @2 line 1
    receiver nil
3 GsNMethod class >> _gsReturnToC         (envId 0) @1 line 1
    receiver nil
topaz 1> temporary
aString abc
```

and to modify the value of the temporary:

```
topaz 1> temporary aString 'xyz'
topaz 1> stack
==> 1 String class >> withAll:            (envId 0) @1 line 1
    receiver String
    aString xyz
2 Executed Code                            @2 line 1
    receiver nil
3 GsNMethod class >> _gsReturnToC         (envId 0) @1 line 1
    receiver nil
```

the method will return the modified value:

```
topaz 1> continue
xyz
```

## THREAD [*anInt*] [CLEAR]

### **thread**

Displays the currently selected GsProcess from among the stack saved from the last error, or from those retrieved by the most recent **threads** command.

### **thread** *anInt*

Changes the currently selected GsProcess. You can specify an integer value from among those shown in the most recent **threads** command.

### **thread** *anInt* **clear**

Clears the selected GsProcess from the Topaz stack cache.

## THREADS [CLEAR]

### **threads**

Forces any dirty instances of GsProcess cached in VM stack memory to be flushed to object memory. Then executes a message send of

**ProcessorScheduler>>topazAllProcesses**

and retrieves and displays the list of processes.

### **threads clear**

Clears the Topaz cache of all instances of GsProcess.

## TIME

The first execution of **time** during the life of a topaz process displays current date and time from the operating system clock, total CPU time used by the topaz process.

Subsequent execution of **time** will display in addition elapsed time since the previous **time** command, CPU time used by the topaz process since the previous **time** command.

The **time** command can be executed when not logged in as well as after login.

```
topaz 1> time
06/08/11 10:06:00.846 PDT
CPU time:  0.050 seconds
topaz 1> run
Array allInstances size
%
18991
topaz 1> time
06/08/11 10:06:09.761 PDT
CPU time:  0.210 seconds
Elapsed Real time:  8.915 seconds
Elapsed CPU  time:  0.160 seconds
```

## **TOPAZPAUSEFORDEBUG [*errorNumber*]**

### **TOPAZWAITFORDEBUG**

These functions are provided to assist in internal debugging of sessions, and are not designed for customer use.

## UNPROTECTMETHODS

Cancels the effect of **protectmethods**, which is used for consistency checking in filein scripts.

—  
|

## UP [*anInteger*]

Moves the scope up within the currently selected call stack, and displays the activation selected as a result. The optional argument *anInteger* specifies how many activations to move up. If no argument is supplied, the scope will go up one activation. See also **stack up** on page 153.

The context displayed includes parameters and temporaries for the context, unlike the results displayed by **stack up**.

```
topaz 1> where
1 AbstractException >> _signalWith: (envId 0) @5 line 25
2 AbstractException >> signal (envId 0) @2 line 47
3 Number >> _errorDivideByZero (envId 0) @6 line 7
==> 4 SmallInteger >> / (envId 0) @6 line 7
5 [] in Executed Code (envId 0) @2 line 1
6 Collection >> do: (envId 0) @5 line 10
7 Executed Code @2 line 1
8 GsNMethod class >> _gsReturnToC (envId 0) @1 line 1
topaz 1> up
5 [] in Executed Code (envId 0) @2 line 1
    self nil
    receiver anExecBlock1
    x 1
topaz 1> where
1 AbstractException >> _signalWith: (envId 0) @5 line 25
2 AbstractException >> signal (envId 0) @2 line 47
3 Number >> _errorDivideByZero (envId 0) @6 line 7
4 SmallInteger >> / (envId 0) @6 line 7
==> 5 [] in Executed Code (envId 0) @2 line 1
6 Collection >> do: (envId 0) @5 line 10
7 Executed Code @2 line 1
8 GsNMethod class >> _gsReturnToC (envId 0) @1 line 1
```

**UPR** [*anInteger*]  
**UPRB** [*anInteger*]

These commands are used with Ruby applications, but not with Smalltalk applications.

—  
|

## WHERE [*anInteger* / *aString*]

Displays the current call stack, with one line per activation.

### **where**

Displays all lines of the current call stack. Equivalent to the **stk** command.

```
topaz 1> where
1 AbstractException >> _signalWith:      (envId 0) @5 line 25
2 AbstractException >> signal            (envId 0) @2 line 47
3 Number >> _errorDivideByZero          (envId 0) @6 line 7
==> 4 SmallInteger >> /                  (envId 0) @6 line 7
5 [] in Executed Code                    (envId 0) @2 line 1
6 Collection >> do:                       (envId 0) @5 line 10
7 Executed Code                           @2 line 1
8 GsNMethod class >> _gsReturnToC       (envId 0) @1 line 1
```

### **where** *anInteger*

Displays the specified number of contexts of the stack, starting with the current active context.

```
topaz 1> where 3
==> 4 SmallInteger >> /                  (envId 0) @6 line 7
5 [] in Executed Code                    (envId 0) @2 line 1
6 Collection >> do:                       (envId 0) @5 line 10
```

### **where** *aString*

Searches all frames in the current stack, and displays only those for which the output of **where** for that frame matches a case-sensitive search for *aString* anywhere in that frame's output (not including the frame number or ==> marker at the start of the frame's line). The current frame is set to the first frame matched by the search.

The string must not begin with a decimal digit, whitespace, or any of the three characters (' + -), and must not contain whitespace. To specify a string that contains digits or whitespace characters, enclose it in single-quotes. For example:

```
where '118 '
```

**WHRB** [*anInteger*]

**WHRUBY** [*anInteger*]

These commands are used with Ruby applications, but not with Smalltalk applications.

—  
|

# Topaz Command-Line Syntax

When Topaz is invoked with the **-l** option, it initiates the program with a linked, as opposed to a remote (RPC) session. Other command-line options give additional control. This section presents the formal command syntax followed by a complete list of command-line options.

## A.1 Command-Line Syntax

By default the **topaz** command invokes an RPC executable. This is the same as specifying the **-r** option on the topaz command line:

```
topaz [ -r ] [ -n netLdiName ] [ -q ] [-I topazini] [ -i ]  
      [ -u useName ] [ -h ]
```

When invoked with the **-l** option, Topaz runs in linked mode. The command line accepts some additional options for the linked version:

```
topaz -l [ -n netLdiName ] [ -e exeConfig ] [ -z systemConfig ]  
      [-T tocSizeKB] [ -q ] [-I topazini] [ -i ] [ -u useName ]  
      [ -h ]
```

## A.2 Options

Arguments are optional. Other than the `-l` option to specify linked topaz, these arguments may not be needed for a standard GemStone configuration.

**-l** Invoke the linked version of Topaz. In this version, Topaz and Gem (the GemStone session) exist as a single process, which significantly enhances performance. The linked version can run only one linked sessions. Any additional sessions are initiated as remote procedure call sessions. If you don't specify this parameter, the remote procedure call version of Topaz is invoked.

If subsequent login parameters set `gemnetid` to a version of `gemnetobject`, topaz will login RPC rather than linked, regardless of the `-l` option.

**-r** Invoke the remote procedure call version of Topaz. In this version, Gems exist as separate processes. If you intend to run multiple GemStone sessions simultaneously, or if you will be running Topaz and your GemStone session on separate nodes, then you must use this version. If you don't specify `-l` or `-r`, Topaz defaults to the remote procedure call version.

**-n** *netLdiName*

The name of the network server process. This may be specified as a GemStone network resource specification. If you don't explicitly specify this parameter, Topaz will look for

- (1) a name specified by the `GEMSTONE_NRS_ALL` environment variable
- (2) a GemStone network server named `gs64ldi`

**-e** *exeConfig*

Executable-specific configuration file. If this argument is not present, the Topaz command uses the customary `GEMSTONE_EXE_CONF` search sequence described in the "Configuration Files" chapter of your *GemStone/S 64 Bit System Administration Guide*. Only applies to linked sessions.

**-z** *systemConfig*

System configuration file. If this argument is not present, the topaz command uses the customary `GEMSTONE_SYS_CONF` search sequence described in the "Configuration Files" chapter of your *GemStone/S 64 Bit System Administration Guide*. Only applies to linked sessions.

**-T** *tocSizeKB*

Sets the `GEM_TEMPOBJ_CACHE_SIZE` that will be used. This overrides any settings provided in configuration files passed as arguments with the `-e` or `-z` options. Only applies to linked sessions.

**-i** Ignore the topaz startup file `.topazini`.

- I** *topazini*  
Specify a complete path and file to a topazini initialization files, and use this rather than any `.topazini` in the default location.
- q** Start Topaz in quiet mode, suppressing printout of the banner and other information.
- u** *useName*  
Pass a string to the topaz executable, which is not used by the executable itself, but may be useful in identifying processes in OS utilities such as `top` or `ps`.
- h** Print a usage line and exit.

—  
|

# *Network Resource String Syntax*

---

This appendix describes the syntax for network resource strings. A network resource string (NRS) provides a means for uniquely identifying a GemStone file or process by specifying its location on the network, its type, and authorization information. GemStone utilities use network resource strings to request services from a NetLDI.

## **B.1 Overview**

One common application of NRS strings is the specification of login parameters for a remote process (RPC) GemStone application. An RPC login typically requires you to specify a GemStone repository monitor and a Gem service on a remote server, using NRS strings that include the remote server's hostname. For example, to log in from Topaz to a Stone process called "gs64stone" running on node "handel", you would specify two NRS strings:

```
topaz> set gemstone !@handel!gs64stone
topaz> set gemnetid !@handel!gemnetobject
```

Many GemStone processes use network resource strings, so the strings show up in places where command arguments are recorded, such as the GemStone log file. Looking at log messages will show you the way an NRS works. For example:

```
Opening transaction log file for read,  
filename = !@oboe#dbf!/user1/gemstone/data/tranlog0.dbf
```

An NRS can contain spaces and special characters. On heterogeneous network systems, you need to keep in mind that the various UNIX shells have their own rules for interpreting these characters. If you have a problem getting a command to work with an NRS as part of the command line, check the syntax of the NRS recorded in the log file. It may be that the shell didn't expand the string as you expected.

#### NOTE

*Before you begin using network resource strings, make sure you understand the behavior of the software that will process the command.*

See each operating system's documentation for a full discussion of its own rules about escaping certain characters in NRS strings that are entered at a command prompt.

If there is a space in the NRS, you can replace the space with a colon (:), or you can enclose the string in quotes (" "). For example, the following network resource strings are equivalent:

```
% waitstone !@oboe#auth:user@password!gs64stone  
% waitstone "!@oboe#auth user@password!gs64stone"
```

## B.2 Defaults

The following items uniquely identify a network resource:

- communications protocol – such as TCP/IP
- destination node – the host that has the resource
- authentication of the user – such as a system authorization code
- resource type – such as server, database extent, or task
- environment – such as a NetLDI, a directory, or the name of a log file
- resource name – the name of the specific resource being requested.

A network resource string can include some or all of this information. In most cases, you need not fill in all of the fields in a network resource string. The

information required depends upon the nature of the utility being executed and the task to be accomplished. Most GemStone utilities provide some context-sensitive defaults. For example, the Topaz interface prefixes the name of a Stone process with the **#server** resource identifier.

When a utility needs a value for which it does not have a built-in default, it relies on the system-wide defaults described in the syntax productions in “Syntax” on page 180. You can supply your own default values for NRS modifiers by defining an environment variable named GEMSTONE\_NRS\_ALL in the form of the *nrs-header* production described in the Syntax section. If GEMSTONE\_NRS\_ALL defines a value for the desired field, that value is used in place of the system default. (There can be no meaningful default value for “resource name.”)

A GemStone utility picks up the value of GEMSTONE\_NRS\_ALL as it is defined when the utility is started. Subsequent changes to the environment variable are not reflected in the behavior of an already-running utility.

When a client utility submits a request to a NetLDI, the utility uses its own defaults and those gleaned from its environment to build the NRS. After the NRS is submitted to it, the NetLDI then applies additional defaults if needed. Values submitted by the client utility take precedence over those provided by the NetLDI.

## B.3 Notation

Terminal symbols are printed in boldface. They appear in a network resource string as written:

**#server**

Nonterminal symbols are printed in italics. They are defined in terms of terminal symbols and other nonterminal symbols:

*username ::= nrs-identifier*

Items enclosed in square brackets are optional. When they appear, they can appear only one time:

*address-modifier ::= [protocol] [@ node]*

Items enclosed in curly braces are also optional. When they appear, they can appear more than once:

*nrs-header ::= ! [address-modifier] {keyword-modifier} !*

Parentheses and vertical bars denote multiple options. Any single item on the list can be chosen:

*protocol* ::= ( **tcp** | **serial** | **default** )

## B.4 Syntax

*nrs* ::= [*nrs-header*] *nrs-body*

where:

*nrs-header* ::= ! [*address-modifier*] {*keyword-modifier*} [*resource-modifier*]!

All modifiers are optional, and defaults apply if a modifier is omitted. The value of an environment variable can be placed in an NRS by preceding the name of the variable with "\$". If the name needs to be followed by alphanumeric text, then it can be bracketed by "{" and "}". If an environment variable named `f00` exists, then either of the following will cause it to be expanded: `$f00` or `${f00}`. Environment variables are only expanded in the *nrs-header*. The *nrs-body* is never parsed.

*address-modifier* ::= [*protocol*] [*@ node*]

Specifies where the network resource is.

*protocol* ::= ( **tcp** | **serial** | **default** )

Supports heterogeneous connections by predicating address on a network type. If no protocol is specified, `GCI_NET_DEFAULT_PROTOCOL` is used. On UNIX hosts, this default is **tcp**.

*node* ::= *nrs-identifier*

If no node is specified, the current machine's network node name is used. The identifier may also be an Internet-style numeric address. For example:

```
!@120.0.0.4#server!cornerstone
```

*nrs-identifier* ::= *identifier*

Identifiers are runs of characters; the special characters `!`, `#`, `$`, `@`, `^` and white space (blank, tab, newline) must be preceded by a "^". Identifiers are words in the UNIX sense.

*keyword-modifier* ::= ( *authorization-modifier* | *environment-modifier* )

Keyword modifiers may be given in any order. If a keyword modifier is specified more than once, the latter replaces the former. If a keyword modifier takes an argument, then the keyword may be separated from the argument by a space or a colon.

*authorization-modifier* ::= ( (**#auth** | **#encrypted**) [:] *username* [*@ password*] )

**#auth** specifies a valid user on the target network. A valid password is needed only if the resource type requires authentication. **#encrypted** is used by

GemStone utilities. If no authentication information is specified, the system will try to get it from the `.netrc` file. This type of authorization is the default.

*username ::= nrs-identifier*

If no user name is specified, the default is the current user.  
(See the earlier discussion of *nrs-identifier*.)

*password ::= nrs-identifier*

If no password is specified, the system will try to obtain it from the user's `.netrc` file. (See the earlier discussion of *nrs-identifier*.)

*environment-modifier ::= ( #netldi | #dir | #log ) [:] nrs-identifier*

**#netldi** causes the named NetLDI to be used to service the request. If no NetLDI is specified, the default is `gs64ldi`. When you specify the **#netldi** option, the *nrs-identifier* is either the name of a NetLDI service or the port number at which a NetLDI is running.

**#dir** sets the default directory of the network resource. It has no effect if the resource already exists. If a directory is not set, the pattern “%H” (defined below) is used. (See the earlier discussion of *nrs-identifier*.)

**#log sets** the name of the log file of the network resource. It has no effect if the resource already exists. If the log name is a relative path, it is relative to the working directory. If a log name is not set, the pattern “%N%P%M.log” (defined below) is used. (See the earlier discussion of *nrs-identifier*.)

The argument to **#dir** or **#log** can contain patterns that are expanded in the context of the created resource. The following patterns are supported:

%H	home directory
%M	machine's network node name
%N	executable's base name
%P	process pid
%U	user name
%%	%

*resource-modifier ::= ( #server | #spawn | #task | #dbf | #monitor | #file )*

Identifies the intended purpose of the string in the *nrs-body*. An NRS can contain only one resource modifier. The default resource modifier is context sensitive. For instance, if the system expects an NRS for a database file, then the default is **#dbf**.

**#server** directs the NetLDI to search for the network address of a server, such as a Stone or another NetLDI. If successful, it returns the address. The *nrs-body* is a network server name. A successful lookup means only that the service has been defined; it does not indicate whether the service is currently running. A

new process will not be started. (Authorization is needed only if the NetLDI is on a remote node and is running in secure mode.)

**#task** starts a new Gem. The *nrs-body* is a NetLDI service name (such as "gemnetobject"), followed by arguments to the command line. The NetLDI creates the named service by looking first for an entry in `$GEMSTONE/sys/services.dat`, and then in the user's home directory for an executable having that name. The NetLDI returns the network address of the service. (Authorization is needed to create a new process unless the NetLDI is in guest mode.) The **#task** resource modifier is also used internally to create page servers.

**#dbf** is used to access a database file. The *nrs-body* is the file spec of a GemStone database file. The NetLDI creates a page server on the given node to access the database and returns the network address of the page server. (Authorization is needed unless the NetLDI is in guest mode).

**#spawn** is used internally to start the garbage collection and other service Gem processes.

**#monitor** is used internally to start up a shared page cache monitor.

**#file** means the *nrs-body* is the file spec of a file on the given host (not currently implemented).

*nrs-body* ::= unformatted text, to end of string

The *nrs-body* is interpreted according to the context established by the *resource-modifier*. No extended identifier expansion is done in the *nrs-body*, and no special escapes are needed.

---

## Symbols

^ (current class) 42, 62  
! (remark) 135  
\*\* (last result) 42, 62  
! (in NRS) 19, 180  
@ (for OOP literal) 42, 62  
@ (for step point index) 54  
@ (in NRS) 180  
# (in NRS) 180  
==> (indicating active context) 54  
\$ (for character literal) 42

## A

**abort** command 33, 60  
aborting transactions 33  
automatic batch processing 36

## B

batch processing from an input file 36

**begin** command 61  
**break** command 62  
  **classmethod** 62  
  **clear** 53, 63  
  **clear all** 53  
  **delete** 63  
  **delete all** 63  
  **disable all** 63  
  **display** 52, 63  
  **enable** 63  
  **enable all** 63  
  **message** 51  
  **method** 51, 62  
breakpoints 50, 62  
  and special methods 62  
  clearing 53, 63  
  continuing GemStone Smalltalk execution  
    after 68  
  deleting 53  
  listing 52, 63, 107  
  method 51, 62  
  methods that cannot have 62

- setting 51, 62
  - byte objects
    - limiting display of 27, 104
    - storing into with structural access 123
    - structural access 124
  - byte values, displaying 28, 71, 125
- C**
- cacheName
    - System method 144
  - CacheName (status output) 144
  - call stack
    - and GemStone Smalltalk debugging 151
    - displaying contents of active 151
    - examining 53, 151
    - and GemStone Smalltalk debugging 151
    - redefining 154
    - removing 155
  - category
    - current 30, 31
    - listing 32
    - setting the current 144
  - category** command 65
  - characters, Topaz syntax for 42
  - class
    - creating with **set class** command 144
    - current 30, 31
    - filing out 35
    - modifying with **set class** command 144
    - setting the current 144
  - class instances
    - Topaz syntax for 82, 85
  - class methods
    - changing 31
    - compiling 66
    - creating 31, 77
    - editing 32
    - modifying 77
  - classmethod** command 31, 66
  - command-line syntax 173
  - commands
    - abbreviation of 59
  - abort** 33, 60
  - begin** 61
  - break** 62
  - case-sensitivity of 59
  - category** 65
  - classmethod** 66
  - commit** 33, 67
  - continue** 55, 68
  - define** 44, 69
  - disassem** 70
  - display** 28, 71, 125
  - doit** 73
  - down** 74
  - downr** 75
  - downrb** 75
  - edit** 76
  - errorcount** 78
  - exit** 47, 79, 133
  - exitifnoerror** 80
  - expectbug** 81
  - expecterror** 82
  - expectvalue** 85
  - fileout** 35, 87
  - frame** 57, 89
  - fr\_cls** 88
  - gcitrace** 90
  - help** 91
  - hierarchy** 92
  - history** 93
  - iferr** 94
  - iferr\_clear** 95
  - iferr\_list** 96
  - iferror** 97
  - implementors** 98
  - input** 99
  - inspect** 100
  - interp** 101
  - interpenv** 102
  - level** 27, 103
  - limit** 27, 104
  - list** 105
  - listw** 109
  - login** 111

**logout** 47, 113  
**logoutifloggedin** 114  
**lookup** 115  
**method** 31, 117  
**nbresult** 118  
**nbrun** 119  
**nbstep** 120  
**obj1** 121  
**obj1z** 122  
**obj2** 121  
**obj2z** 122  
**object** 123  
**omit** 28, 71, 125  
**output** 127  
**pausefordebug** 129  
**pkglookup** 130  
**printit** 24, 131  
**protectmethods** 132  
**quit** 133  
**releaseall** 134  
**remark** 135  
**removeallclassmethods** 136  
**removeallmethods** 137  
**rubyclassmethod** 138  
**rubyhierarchy** 138  
**rubyimplementors** 138  
**rubylist** 138  
**rubylookup** 138  
**rubymethod** 138  
**rubyrun** 138  
**run** 139  
**runenv** 140  
**send** 46, 141  
**sendenv** 142  
**senders** 143  
**set** 19, 144  
**shell** 149  
**spawn** 150  
**stack** 53, 57, 151  
**status** 24, 31, 156  
**step** 50, 157  
**stk** 53, 158  
**strings** 159

**stringsic** 160  
**subclasses** 161  
 syntax of 59  
**temporary** 56, 162  
**thread** 164  
**threads** 165  
**time** 166  
**topazpausefordebug** 167  
**topazwaitfordebug** 167  
**unprotectmethods** 168  
**up** 169  
**upr** 170  
**uprb** 170  
**where** 53, 171  
**whrb** 172  
**whruby** 172  
 comments 135  
**commit** command 33, 67  
 committing transactions 33  
 compile\_env: 145  
 context  
   and GemStone Smalltalk debugging 151  
   displaying the active 153  
   listing method breakpoints 52, 107  
   listing step points in 106  
   redefining the active 57, 153  
   selecting 57  
**continue** command 55, 68  
 Control-C handling 38  
 current category, setting 65, 144  
 current class  
   and **classmethod** command 66  
   and **method** command 117  
   setting 144

## D

debugging 49–53, 151, 157, 158, 162  
   and execution context 57, 151  
**define** command 44, 69  
**disassem** command 70  
**display** command 28, 71, 125  
   **oops** and stack display 151

display level 26–27  
   maximum 103  
 display of results, controlling 26  
**doit** command 73  
**down** command 74  
**downr** command 75  
**downrb** command 75

## E

**edit** command 76  
   and **set editorname:** command 145  
   **classmethod** 32, 77  
   **last** 25, 76  
   **method** 32, 77  
   **new classmethod** 32, 77  
   **new method** 32, 76  
   **new text** 25  
   used with **set** command 144  
 editing GemStone Smalltalk expressions 25  
 environmentId 16, 145, 151  
 error status 79, 133  
**errorcount** command 78  
 errors, continuing GemStone Smalltalk  
   execution after 68  
 examining the call stack 53  
 execution, stepping through 50, 106  
**exit** command 47, 79, 133  
**exitifnoerror** command 80  
**expectbug** command 81  
**expecterror** command 82  
**expectvalue** command 85

## F

file  
   appending to 33  
   input 36, 37  
   output 33, 37  
   redirection 33  
**fileout** command 35, 87  
 finding method in hierarchy 115  
 Floats, Topaz syntax for 42

**frame** command 57, 89  
**fr\_cls** command 88  
 ftplogin. 23

## G

**gcitrace** command 90  
 gemnetobject 146  
 GemStone  
   aborting a transaction 60  
   call stack and GemStone Smalltalk  
     debugging 151  
   committing a transaction 67  
   context and error handling 151  
   examining the call stack 151  
   interrupting 38  
   logging in 18  
   logging out 47  
   multiple sessions 38, 148  
   service, setting 145  
 GemStone name 18, 146  
   setting 111  
 GemStone password 19  
   setting 111, 148  
 GemStone service name, setting 111  
 GemStone Smalltalk  
   breakpoints 62, 107  
   continuing execution 55, 68  
   debugging 49–53, 151, 157, 158, 162  
   editing expressions 25  
   editing source code 76, 145  
   executing expressions 24, 73, 101, 131, 139  
   sending messages 141  
 GemStone username 19  
   setting 111, 148

## H

**help** command 24, 91  
 hexadecimal values, displaying 28, 71, 125  
**hierarchy** command 92  
**history** command 93  
 history, setting size using **set history** 146

host password 19, 146  
  setting 111  
host username 19, 147

## I

**iferr** command 94  
**iferr\_clear** command 95  
**iferr\_list** command 96  
**iferror** command 97  
**implementors** command 98  
initialization file  
  and **set host password** command 146  
  and **set password** command 148  
  used to set login parameters 111  
**input** command 37, 99  
  **pop** 99  
**inspect** command 100  
instance methods  
  compiling 117  
  creating 76  
  modifying 77  
instance variables  
  displaying 26, 28, 71, 103, 125  
  returning the values of 123  
instances of a class, Topaz syntax for 82, 85  
integers, Topaz syntax for 42  
**interp** command 101  
**interpenv** command 102  
interrupting execution 38

## L

**level** command 27, 103  
**limit** command 27, 104  
  **bytes** 104  
  **oops** 104  
**list** command 105  
  **breaks** 52, 107  
  **breaks classmethod** 108  
  **breaks method** 108  
  **classmethod** 32  
  **method** 32

**steps** 106  
  **steps classmethod** 107  
  **steps method** 50, 107  
**listw** command 109  
listwindow  
  147  
logging a session 37  
logging in to GemStone 18, 111  
**login** command 111  
login initialization file 22  
login parameters 18–23  
  and **set** command 144  
  displaying the value of 156  
**logout** command 47, 113  
**logoutifloggedin** command 114  
**lookup** command 115

## M

message breakpoints  
  listing 52, 63  
  setting 51  
method breakpoints 51, 62  
  listing 52, 63, 107  
  setting 51, 62  
**method** command 31, 117  
method compilations and **set category**  
  command 144  
methods  
  compilation 117  
  compilation and current category 65  
  creating 30, 76  
  editing 32  
  examining and modifying arguments 56  
  filing out 35  
  finding in hierarchy 115  
  listing 32, 115  
  modifying 30  
  stepping through execution 106, 157  
multiple sessions 38, 148

**N**

**nbresult** command 118  
**nbrun** command 119  
**nbstep** command 120  
 .netrc 23, 147  
 network  
   resource string syntax 177  
 network communications and login  
   parameters 111  
 network initialization file 23, 147  
 network resource string (NRS) 111  
 network server process, establishing the name  
   of 145  
 nonsequenceable collections (NSCs)  
   structural access 123  
 NRS (network resource string)  
   syntax 177  
 nrsdefaults 147

**O**

**obj1** command 121  
**obj1z** command 122  
**obj2** command 121  
**obj2z** command 122  
**object** command 123  
   **at:** 41, 123  
   **at:put:** 123, 124  
 object headers 29  
 objects, syntax for specifying 42  
**omit** command 28, 71, 125  
   **oops**, and stack display 151  
 OOPs  
   displaying 28, 71, 125  
   limiting display of 104  
   Topaz syntax for 42, 62  
 operating system error status 79, 133  
**output** command 127  
   and host environment names 127  
   **pop** 127  
   **push** 33, 127  
 output to a file 33

**P**

padding for stack display, using **set stackpad**  
 148  
 password  
   GemStone 19, 148  
   host 19, 146  
 pause message, continuing GemStone  
   Smalltalk execution after 68  
**pausefordebug** command 129  
**pkglookup** command 130  
 prerequisites 3  
**printit** command 24, 131  
   editing the text of 76  
**protectmethods** command 132

**Q**

**quit** command 133  
 quitting Topaz 47

**R**

recording a session 37  
**releaseall** command 134  
**remark** command 135  
**removeallclassmethods** command 136  
**removeallmethods** command 137  
 Ruby  
   and GemStone Smalltalk 16  
**rubyclassmethod** command 138  
**rubyhierarchy** command 138  
**rubyimplementors** command 138  
**rubylist** command 138  
**rubylookup** command 138  
**rubymethod** command 138  
**rubyrun** command 138  
**run** command 139  
**runenv** command 140

**S**

**send** command 46, 141  
**sendenv** command 142

- senders** command 143
  - service name, GemStone 145
  - session numbers 39, 148
  - sessions, multiple 38, 148
  - set** command 19, 144
    - cachename** 144
    - category** 144
    - class** 30, 144
      - and **edit classmethod** command 77
      - and **edit method** command 77
      - and **edit new classmethod** command 77
      - and **edit new method** command 76
    - compile\_env:** 145
    - editorname** 25, 76
    - editorname:** 145
    - establishing login parameters 111, 144
    - gemnetid** 145
    - gemstone** 146
    - hostpassword** 146
    - hostusername** 147
    - listwindow** 147
    - nrsdefaults** 147
    - password** 148
    - session** 38, 148
      - and local variables 69
    - sourcestringclass** 148
    - stackpad** 148
    - username** 148
  - setcommand**
    - history** 146
  - shell** command 149
  - spawn** command 150
  - special methods
    - and breakpoints 62
  - stack** command 53, 151
    - all** 154
    - change** 154
    - delete** 155
    - nosave** 154
    - save** 154
    - scope** 57, 153
  - stackpad 148
  - standard input, redirecting 37
  - standard output, redirecting 33
  - status** command 24, 31, 156
  - stdin 33
  - stdout 33
  - step** command 50, 157
    - into** 55, 157
    - over** 55, 157
  - step points 50
    - examining 50, 106
    - methods that have no 62
  - stk** command 53, 158
  - stopping execution 38
  - Strings
    - limiting display of 27, 104
    - Topaz syntax for 42
  - strings** command 159
  - stringsic** command 160
  - structural access 40–123
    - and **object** command 123
  - subclasses** command 161
  - Symbols, Topaz syntax for 42
- ## T
- temporaries, examining and modifying 56, 162
  - temporary** command 56, 162
  - thread** command 164
  - threads** command 165
  - time** command 166
  - Topaz
    - command-line syntax 173
    - exiting 47
    - initialization file 22
    - interrupting 38
    - invoking 16
    - linked version 18

- redirecting input 99
- redirecting output 127
- RPC version 18
- syntax for characters 42
- syntax for commands 59
- syntax for Floats 42
- syntax for instances of a class 82, 85
- syntax for integers 42
- syntax for literals 42
- syntax for OOPs 42, 62
- syntax for Strings 42
- syntax for Symbols 42
- syntax for variable names 42, 62
- Topaz initialization file
  - and **set host password** command 146
  - and **set password** command 148
- Topaz variables
  - and **define** command 69
  - and **object** command 123
- topaz.ini 22
- topazini.tpz 22
- topazpausefordebug** command 167
- topazwaitfordebug** command 167
- transactions, aborting 33, 60
- transactions, committing 33

## U

- unprotectmethods** command 168
- up** command 169
- upr** command 170
- uprb** command 170
- username
  - GemStone 19, 148
  - host 19, 147

## V

- variable names, Topaz syntax for 42, 62
- variables, local 44
  - and **define** command 69
  - and **object** command 123
  - clearing definition of 46

- variables, predefined
  - CurrentCategory 45
  - CurrentClass 45
  - ErrorContext 45
  - LastResult 45
  - LastText 45, 76
  - myUserProfile 45

## W

- where** command 53, 171
- whrb** command 172
- whruby** command 172
- writing to a file 87