Topaz User's Guide for GemStone/S 64 Bit™

Version 3.7

September 2023

GEMTALK™ SYSTEMS

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

Preface

About This Manual

This manual describes Topaz, the command-line interface for GemStone/S 64 Bit™. You can use Topaz with the other GemStone development tools to build and maintain comprehensive database applications.

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

Terminology Conventions

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

Technical Support

Support Website

gemtalksystems.com

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

- **Documentation** for the current and for previous released versions of all GemTalk products, in PDF form.
- **Product download** for the current and selected recent versions of GemTalk software.

- **Bugnotes**, identifying performance issues or error conditions that you may encounter when using a GemTalk product.
- **Supplemental Documatation** and **TechTips**, providing information and instructions that are not in the regular documentation.
- **Compatibility matrices**, listing supported platforms for GemTalk product versions.

We recommend checking this site on a regular basis for the latest updates.

Help Requests

GemTalk Technical Support is limited to customers with current support contracts. Requests for technical assistance may be submitted online (including by email), or by telephone. We recommend you use telephone contact only for urgent requests that require immediate evaluation, such as a production system down. The support website is the preferred way to contact Technical Support.

Website: techsupport.gemtalksystems.com

Email: techsupport@gemtalksystems.com

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

Please include the following, in addition to a description of the issue:

- The versions of GemStone/S 64 Bit and of all related GemTalk products, and of any other related products, such as client Smalltalk products, and the operating system and version you are using.
- Exact error message received, if any, including log files and statmonitor data if appropriate.

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

24x7 Emergency Technical Support

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

Training and Consulting

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

Table of Contents

Chapter 1. Getting Started with Topaz 11

1.1 Getting started with Topaz . . . 11
- Overview of a GemStone Session . . . 12
- Remote Versus Linked Versions . . . 12
- Invoking Topaz . . . 13
- Topaz Commands . . . 13
- Logging In to GemStone . . . 14
  - Logging In Linked . . . 15
  - Logging In RPC . . . 16
- Setting Up a Login Initialization File .topazini . . . 18
  - Error handling and output . . . 18
  - Alternatives to automatic initialization . . . 19
  - Special care needed when setting gemnetid in .topazini . . . 19
- Multiple Concurrent GemStone Sessions . . . 19
  - Multiple sessions in the RPC version of Topaz . . . 20
  - Multiple sessions in the Linked version of Topaz . . . 20
  - Topaz sessions vs. GemStone sessions . . . 21
- Transaction state . . . 21
- Other Types of Logins . . . 22
  - X509-Secured . . . 22
  - Solo Logins . . . 22
  - Object creation and memory use . . . 23
  - Solo login with external connection to a running stone . . . 23
- Multiple Execution Environments . . . 24

1.2 Interacting with Topaz . . . 24
- Help Command. . . . 24
- Interrupting Topaz and GemStone . . . 24
- Logging Out. . . . 25
- Leaving Topaz . . . 25

1.3 Executing GemStone Smalltalk Expressions . . . 25
Strings vs. Unicode strings . . . 26
Controlling the Display of Results . . . 26
Display Level. . . . 26
Setting Limits on Object Displays . . . 28
Displaying Variable Names, OOPs, and Byte Values . . . 28
Committing and Aborting Transactions . . . 29
Importing files: topaz commands and GemStone code . . . 29
Handling text outside the ASCII range . . . 30
Capturing Your Topaz Session In a File . . . 31
Writing to multiple log files . . . 31
Invoking Operating System Functionality from Topaz . . . 32
1.4 Using Topaz for Code Development . . . 33
Creating Methods . . . 33
Using a Text Editor to Edit Methods . . . 34
Listing Methods, Categories, and other information . . . 35
Filing Out Classes and Methods . . . 36
Code outside the ASCII range . . . 37
1.5 Advanced Topaz features . . . 37
Structural Access To Objects . . . 37
Examining Instance Variables with Structural Access . . . 38
Specifying Objects . . . 38
Object Specification Formats . . . 38
Topaz Variables . . . 39
Creating Variables . . . 40
Displaying Current Variable Definitions . . . 40
Clearing Variable Definitions . . . 41
Sending Messages . . . 41

Chapter 2. Scripting with Topaz and SuperDoit 43

Standard and Solo logins . . . 43
2.1 Topaz scripting . . . 44
Creating a script . . . 44
Security Concerns . . . 45
2.2 Scripting with topaz she-bang . . . 46
2.3 Script Arguments . . . 47
Passing arguments using bash scripts . . . 47
2.4 SuperDoit . . . 49
Script Types . . . 49
Script structure . . . 49
Commonly Used Sections . . . 50
Complex Scripting . . . 51
Rowan Support . . . 51
Script Comments . . . 51
Arguments . . . 51

Standard arguments . . . 52
Defining Script Arguments . . . 52
Arguments to Topaz . . . 53
Usage . . . 53
Coding in the doit section . . . 53
Output . . . 54
Examples and Templates . . . 54
$GEMSTONE/examples/superDoit/ . . . 54
Rowan-specific GsCommands examples . . . 54

Chapter 3. Debugging Your GemStone Smalltalk Code 55

3.1 Step Points and Breakpoints . . . 55
Breakpoints . . . 56
3.2 Examining the GemStone Smalltalk Call Stack . . . 57
Proceeding After a Breakpoint . . . 59
Select a Context for Examination and Debugging . . . 60
Multiple Call Stacks . . . 60
3.3 Debugging from a different session . . . 62
waitForDebug . . . 62
Debugging . . . 63
Manual setup of remote debugging . . . 63
Enabling the debugging thread . . . 64
Determining the OOP of executing session's process . . . 64
Executing session waiting for debug . . . 64
Example . . . 64

Chapter 4. Command Dictionary 67

ABORT . . . 68
ALLSTACKS . . . 69
BEGIN . . . 70
BREAK . . . 71
CATEGORY . . . 74
CLASSMETHOD . . . 75
COMMIT . . . 76
CONTINUE/C . . . 77
DEBUGGEM . . . 78
DEBUGRUN . . . 80
DEFINE . . . 81
DETACH . . . 82
DISASSEM . . . 83
DISPLAY . . . 84
DOIT . . . 88
DOWN . . . 89
DUMPOBJ . . . 90
EDIT . . . 91

ENV 93
ERRORCOUNT 94
EXEC 95
EXIT 96
EXITIFNOERROR. 97
EXPECTBUG 98
EXPECTERROR 99
EXPECTVALUE. 102
FILEFORMAT 104
FILEOUT. 105
FR_1 106
FR_CLS. 107
FRAME. 108
GCITRACE 109
HELP 110
HIERARCHY 111
HISTORY 112
IFERR. 113
IFERR_CLEAR 115
IFERR_LIST 116
IFERROR. 117
IMPLEMENTORS. 118
INPUT 119
INSPECT 120
INTERP 121
KILL 122
LEVEL *123*
LIMIT. 124
LIST. 125
LISTW / L 130
LITERALS 131
LOADUA 132
LOGIN 133
LOGOUT. 134
LOGOUTIFLOGGEDIN 135
LOOKUP. 136
METHOD 139
NBRESULT 140
NBRUN 141
NBSTEP 142
OBJ1 / OBJ2 143
OBJ1Z / OBJ2Z 144
OBJECT 145
OMIT 147
OUTPUT 150
PAUSEFORDEBUG. 152
PKGLOOKUP 153
POLLFORSIGNAL 154
PRINTIT 155
PROTECTMETHODS 156
QUIT 157

RELEASEALL 158
REMARK 159
REMOVEALLCLASSMETHODS. 160
REMOVEALLMETHODS 161
RESUME 162
RUN 163
RUNBLOCK 164
RUNENV *envId* *165*
SEND 166
SENDERS 167
SET. 168
SHELL 176
STACK 177
STATUS 181
STEP 182
STK 183
STRINGS 184
STRINGSIC 185
SUBCLASSES. 186
SUBHIERARCHY 187
TEMPORARY. 188
TFILE 190
THREAD 191
THREADS. 192
TIME. 193
TMETHOD 194
TOPAZWAITFORDEBUG 195
UNPROTECTMETHODS 196
UP 197
WHERE 198

Appendix A. Topaz Command-Line Syntax 199

Chapter 1 Getting Started with Topaz

Topaz is a linear interface for GemStone/S 64 Bit™ that provides scripting and command-line access to the GemStone system. Topaz does not require a windowing system, and so is a useful interface 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.

To run Topaz, GemStone/S 64 Bit must be installed on your system. You must have an running repository monitor (Stone) that is the same version of GemStone as Topaz, and in some cases an accessible network service process (NetLDI). The *GemStone/S 64 Bit Installation Guide* explains how to install these components.

In most cases, your environment should contain a definition of the `$GEMSTONE` environment variable and your execution path should include the GemStone binary directory `$GEMSTONE/bin`.

Examples throughout this book were created on a UNIX system. Topaz is also available with the GemStone/S 64 Bit Windows Client distribution, which allows Topaz to run on Windows, logging in remotely to a GemStone server running on UNIX. Topaz on Windows cannot login in linked mode, nor with a Gem session on Windows. Otherwise, Topaz operates similarly on UNIX and Windows. Differences are noted in the text.

1.1 Getting started with Topaz

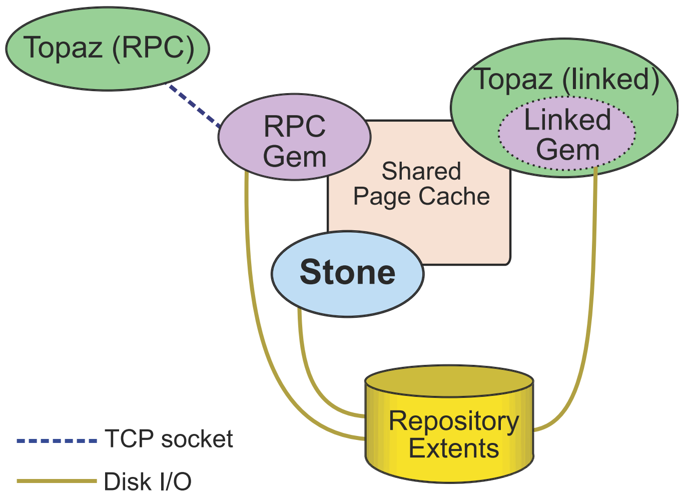
Topaz provides an environment in which you can login a GemStone session and use that session to find out information about GemStone, and to execute Smalltalk code. To perform any server interaction, you must first log in. All server interaction is performed using the logged-in session. The sessions that Topaz uses are the same as sessions acquired via any other GemStone interface, such as GemBuilder for Smalltalk (GBS).

Overview of a GemStone Session

A GemStone session consists of four parts, as shown in Figure 1.1. These are:

- An **application**, in this case, Topaz.
- One **repository** and repository monitor (**Stone**), and the associated server processes. An application has one repository to hold its persistent objects.
- 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 1.1 GemStone Object Server Components



Remote Versus Linked Versions

Figure 1.1 includes the two types of topaz sessions -- *Remote Procedure Call* (*RPC*) and *Linked*. With a Topaz RPC login, the Topaz application and the Gem session that provides the repository services exist in two separate processes. In a linked login, the Topaz application process includes the Gem session. These two types of sessions and configuration details are described in more detail in the *System Administration Guide for GemStone/S 64 Bit*.

To allow control over the use of linked logins, GemStone client libraries and Topaz provide separate versions; one which allows RPC logins only, the other allowing Linked and RPC.

- Topaz RPC is the default, although you may specify the **-r** command line argument. You can run multiple RPC sessions from the Topaz RPC executable, but linked sessions are not possible.
- Topaz Linked requires specifying the **-L** or **-l** command line argument to topaz. You can run a single linked session and multiple RPC sessions from the linked topaz executable.

In Figure 1.2, notice that the Topaz startup banner's `PROGRAM` line refers to `Remote Session`. In a linked login, it will say `Linked Session`.

Under Windows, only the RPC version of Topaz is available. The Gem, which is part of the server code, can only 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.

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 1.2.

Figure 1.2 Topaz Banner and Prompt

```
unix> topaz
 _______________________________________________________________________________
|             GemStone/S64 Object-Oriented Data Management System               |
|               Copyright (C) GemTalk Systems 1986-2019                         |
|                            All rights reserved.                               |
+-------------------------------------------------------------------------------+
|    PROGRAM: topaz, Linear GemStone Interface (Remote Session)                 |
|    VERSION: 3.7.0, Fri Aug 11 15:04:14 2023                                   |
|     COMMIT: 2023-08-11T15:39:57-07:00 0b86af1699855748c22a91533ad52e62fa69da1|
|  BUILT FOR: x86-64 (Linux)                                                    |
| RUNNING ON: 8-CPU lark x86_64 (Linux 5.4.0-135-generic #152-Ubuntu SMP Wed   |
| Nov 3420:02:41 UTC 2022)                                                      |
|  PROCESSOR: 8-core AMD Ryzen 9 5900X 12-Core Processor  (Bulldozer)           |
|     MEMORY: 32112 MB                                                          |
| PROCESS ID: 2594932    DATE: 08/14/2023 10:19:50 (UTC -7:00)                  |
|   USER IDS: REAL=gsuser (534) EFFECTIVE=gsuser (534) LOGIN=gsuser (534)       |
|_______________________________________________________________________________|
topaz>
```

Topaz Commands

Topaz interaction is performed by sending commands. Each command begins with a keyword, and may have optional or required arguments. Some commands are only meaningful in certain contexts. Chapter 4 provides a list of the commands you will be using and the options and arguments for each command.

The first thing you will do is use the **set** command to provide login parameters, and use the **login** command to perform the login to GemStone. This is described in more detail in the next section.

The commands and keywords in Topaz are case-insensitive; so for example, the command **login** can be entered as LOGIN or Login. Arguments to topaz commands, however, such as user names or the names of Smalltalk classes, will follow their specific rules for case sensitivity.

You can abbreviate many Topaz command to uniqueness, so for example you can type **log** instead of **login**. Some commands such as **logout** may not be abbreviated, to avoid accidental use. The colon indicating arguments is often omitted, and the examples in this manual generally do not include the colon.

To enter an argument to a Topaz command that includes white space, enclose it within single quotes. For example, a username such as 'Issac Newton' requires quotes, while Issac_Newton does not.

Normally, each topaz command is on a separate line, and the command is terminated by the return at the end of the line. For most commands, you can include multiple topaz commands on a single line by separating each expression with a semicolon (;), however, commands that interpret the rest of the line in specific ways, such as run, do not allow this usage. Commands may not be longer than 64K characters.

Any text following a command that does not have arguments, or text following the argument, is ignored; topaz will print a warning to that effect after executing the valid portion of the line.

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 the Stone's name, and a valid GemStone user name and password.

Note
This section does not apply to X509-Secured GemStone, which takes a completely different set of login parameters. Logins using X509-Secured GemStone are described in the GemStone/S 64 Bit X509-Secured GemStone Administration Guide.

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

- **GemStone name**. This the name of the Stone process to login into, optionally including the NRS information required to locate the Stone.

  If the node where the Stone is running is not the same one on which the Gem will run, you also need the name of the Stone host. For example, to specify a process named `gs64stone` running on node `central`, you can use a network resource string of the form `!@central!gs64stone`.

  For details on using NRS and NRS syntax, see the *System Administration Guide for GemStone/S 64 Bit*.

  This is configured using the `set` command: `set gemstone` *stonename.*

- **GemStone user name and password**. These are defined within the GemStone server. You can log in using your personal username and password, created by your GemStone Administrator, or as predefined GemStone system users such as DataCurator.

  These are configured using the `set` command, `set username` *gemStoneUserName* and `set password` *gemStoneUserPassword*. You may abbreviate and combine them as `set user` *gemStoneUserName* `pass` *gemStoneUserPassword.*

  You may omit the argument to the `set password` command, in which case you will be prompted to enter the password.

The default password for predefined GemStone users such as DataCurator is `swordfish`. However, it is strongly recommended that this is changed, to provide some basic system security.

- **host user name and password**. The UNIX name and password for the account on the server that will own the Gem process. These are needed only for RPC sessions, and only if the NetLDI is configured to require this for authentication. More information on NetLDI modes can be found in the *System Administration Guide for GemStone/S 64 Bit*.

  These are configured using the `set` command, `set hostusername` *osUserName* and `set hostpassword` *osPassword*. You may abbreviate and combine them as `set hostuser` *osUserName* `hostpass` *osPassword.*

  You may omit the argument to the `set hostpassword` command, in which case you will be prompted to enter the password.

- **GemStone service name**. For the RPC version the default is `gemnetobject`. You may also use `gemnetdebug`, if you are debugging memory issues, or the name of a customized version of the `gemnetobject` script that you have created.

  For the linked version of Topaz, do **not** set gemnetid, nor set the gemnetid to ''.

  NOTE
  If you are running the linked version of topaz, and set gemnetid to `gemnetobject`*, all your sessions will be RPC, in spite of having invoked the linked version. Use -L to start linked topaz to avoid this issue.*

  The GemStone service name is configured using the `set` command `set gemnetid` *serviceName.* The gem service can be provided an NRS, in order to specify that the Gem process is running on a remote node. To do this, for the gemnetid, specify a network resource string (NRS) of the form `!@<`*remoteNode*`>#netldi:<`*netldiName*`>!gemnetobject`.

  For example,

  `!@lark.gemtalksystems.com#netldi:gs64ldi!gemnetobject`.

  Additional NRS directives and gemnetobject arguments can be included. For details on using NRS and NRS syntax and the `gemnetobject` utility, see the *System Administration Guide for GemStone/S 64 Bit*.

Logging In Linked

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

```
topaz> set gemstone gs64stone
topaz> set username Isaac_Newton
topaz> set password
GemStone Password?  (type the GemStone password for Issac_Newton)
```

You may supply several of these login parameters on a single command line in any order. except that a password must follow the user name that it is associated with; and parameter names may be abbreviated. Instead of the above three statements, you could just enter:

```
topaz> set gemstone gs64stone user Isaac_Newton pass gravity
```

When logging in linked, do **not** set the **gemnetid** parameter. Any values set for **hostusername** and **hostpassword** are ignored.

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

```
topaz> login
[Info]: LNK client/gem GCI levels = 37000/37000
--- 08/08/2023 11:51:22.943 PDT Login
[Info]: User ID: DataCurator
[Info]: Repository: gs64stone
[Info]: Session ID: 5 login at 08/08/2023 11:51:22.948 PDT
[Info]: GCI Client Host: <Linked>
[Info]: Page server PID: -1
[Info]: using libicu version 58.2
[08/08/2023 11:51:22.951 PDT]
  gci login: currSession 1  linked session
successful login
topaz 1>
```

Logging In RPC

For RPC login, there are additional parameters you may or may not need to specify, depending on how the NetLDI is set up.

The Stone name and the username and password are needed:

```
topaz> set gemstone gs64stone
topaz> set username Isaac_Newton
topaz> set password
GemStone Password? (type the GemStone password for Issac_Newton)
```

If your NetLDI requires host authentication, you will need to provide the host login credentials. These settings can be skipped if the NetLDI is in guest mode.

```
topaz> set hostusername newtoni
topaz> set hostpassword
Host Password? (type the unix password for newtoni)
```

When you are running the RPC version of Topaz, by default **gemnetid** is set to `gemnetobject`, which is sufficient for login when the Stone, NetLDI and Gem are all running on the same node. If the Stone is running on another node, and the NetLDI is not named gs64ldi, you may need to include further details:

```
topaz> set gemnetid !@lark#netldi:54321!gemnetobject
```

You are now ready to issue the **login** command:

```
topaz> login
[08/08/2023 11:44:05.849 PDT]
  gci login: currSession 1  rpc gem processId 20964 socket 6
successful login
topaz 1>
```

Topaz displays a session number in its prompt once you have logged in. In topaz RPC, where multiple logins are allowed, the prompt will indicate which session is the current one.

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

```
topaz 1> status

Current settings are:
 display : 0
 byte limit: 0 lev1bytes: 100
 omit bytes
 include deprecated methods in lists of methods
 display instance variable names
 display oops   omit alloops   omit stacktemps
 oop limit: 0
 omit automatic result checks
 omit interactive pause on errors
 omit interactive pause on warnings
 listwindow: 20
 stackpad: 45 singlecolumn: Off  tab (ctl-H) equals 8 spaces when
   listing method source
 transactionmode  autoBegin
 using line editor
   line editor history: 100
   topaz input is from a tty on stdin
EditorName________ vi
CompilationEnv____ 0
Source String Class String
fileformat          utf8
SessionInit         On
EnableRemoveAll     On
CacheName_________ 'TopazR'

Connection Information:
UserName___________ 'Isaac_Newton'
Password __________ (set)
HostUserName_______ 'newtoni'
HostPassword_______ (set)
NRSdefaults________ '#netldi:gs64ldi'
GemStone___________ 'gs64stone'
GemStone NRS_______ '!#encrypted:newtoni@password#server!gs64stone'
GemNetId___________ 'gemnetobject'
GemNetId NRS_______ '!#encrypted:newtoni@password!gemnetobject'

Browsing Information:
Class______________
Category___________ (as yet unclassified)
```

Setting Up a Login Initialization File .topazini

You can streamline the login process by creating an initialization file that contains the **set** commands needed for logging in. If the file uses the default name, and it is located in one of the search locations, the commands in this file are automatically executed when topaz starts up. If you insert **set password** and **hostpassword** commands without parameters, Topaz automatically prompts you for the necessary values.

Table 1.1 Topaz Initialization File Names

Platform	Name of Topaz Initialization File	Locations searched for initialization file
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 `C:\users\default`.

You may also explicitly specify a path for a topazini file on the command line where you started up the Topaz executable; this file does not need to use the default name. Using this option overrides any default topazini files that Topaz would otherwise use.

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

If you want to run Topaz non-interactively, that is for automated scripts, you must explicitly specify both the GemStone and host passwords in this initialization file. **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 username Isaac_Newton
set password gravity
set hostusername 'newtoni'
set hostpassword calculus
login
```

If you choose not to include your password in an initialization file, Topaz will start up with prompts, for example.

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

Error handling and output

Commands that are executed from a login initialization file are not echoed to the display. However, if an error occurs, the output is reported to the topaz display, so you can determine the cause of the problem. In this case, the password and host password are struck out, for security.

Alternatives to automatic initialization

If a topaz initialization file exists in one of the search directories, it will be executed each time topaz is started. You can prevent this by starting Topaz using the **-i** argument, which suppresses loading of an initialization file.

You can also explicitly provide the initialization file to use on the topaz command line, by using the **-I** command, passing in the name of the initialization file. If you name your file other than the default, or locate this file in a directory other than the ones searched for, you can control the specific one you wish to load.

This also allows you to have multiple initialization files for different uses.

See Appendix A for more details on command line options.

Special care needed when setting gemnetid in .topazini

Use caution in including commands to set the gemnetid in a .topazini file; for example, using a line such as:

```
set gemnetid gemnetobject
```

The `.topazini` file is executed both for RPC topaz, and for linked topaz started using the **-l** option. If this line is present in a `.topazini` file, and that `.topazini` file is executed by **topaz -l**, the effect is to turn the linked login into an RPC login.

When linked topaz is started using the **-L** option, **set gemnetid** commands in `.topazini` files are ignored. For this reasons, using **-L** rather than **-l** is recommended. Otherwise there is no difference between starting linked topaz using **topaz -l** vs. **topaz -L**.

For more on the behavior of gemnetid in linked and RPC topaz, see under **gemnetid** starting on page 170.

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. While you can login both RPC and linked session from the linked version of topaz, the RPC version of topaz does not allow linked sessions.

Multiple sessions in the RPC version of Topaz

The following example shows how you might login a second session to a different GemStone server, make the new session your current session, then return to the original session. With the RPC version of Topaz, all sessions are always RPC.

```
topaz> login
[08/08/2023 12:34:02.292 PDT]
 gci login: currSession 1 rpc gem processId 95
successful login
topaz 1> set gemstone !@srv2!gs64stone
topaz 1> set username Isaac_Newton
Warning: GemStone is clearing previous GemStone password.
GemStone password? <password typed here but not echoed>
topaz 1> login
[08/08/2023 12:34:05.548 PDT]
 gci login: currSession 2 rpc gem processId 141
successful login
topaz 2> exec UserGlobals at: #myVar put: 1 %
1
topaz 2> set session 1
topaz 1>
```

Multiple sessions in the Linked version of Topaz

If you use the **topaz -L** or **-l** command to invoke Topaz, you may run multiple sessions, but only one of them may be linked. Normally, this is the first session that logs in. In order to run two sessions, you must specify a value for gemnetid, and provide OS credentials if your server configuration requires this, so you can get an RPC login.

If you are running linked topaz, and you first login an RPC session, then wish to login the linked session, you will need to clear the setting for gemnetid. For example:

```
topaz> set gemnetid gemnetobject
topaz> login
[08/08/2023 14:40:49.338 PDT]
  gci login: currSession 1  rpc gem processId 31427 socket 6
successful login
topaz 2> set gemnetid ''
topaz 2> login
[Info]: LNK client/gem GCI levels = 37000/37000
--- 08/08/2023 14:37:56.603 PDT Login
[Info]: User ID: DataCurator
[Info]: Repository: gs64stone
[Info]: Session ID: 5
[Info]: GCI Client Host: <Linked>
[Info]: Page server PID: -1
[08/08/2023 14:37:56.609 PDT]
  gci login: currSession 1  linked session
successful login
topaz 1>
```

The messages displayed during login indicate if the session is linked or RPC.

Topaz sessions vs. GemStone sessions

Notice that the Topaz prompt always shows the number of the current session. These sequential session numbers are assigned by your individual Topaz environment, and do not correspond to the session numbers that are assigned by the GemStone server to each logged-in 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: GcUser
session number: 4 UserId: SymbolUser
session number: 5 UserId: DataCurator
session number: 6 UserId: Isaac_Newton
session number: 7 UserId: Isaac_Newton
session number: 8 UserId: Gottfried_Leibniz
```

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 canonicality.

This list includes all sessions that are currently logged into the GemStone system, not only the sessions within your Topaz environment.

Transaction state

GemStone is a transactional system. The process of making persistent changes in the GemStone repository require that you begin a transaction, make the changes, then commit the transaction.

To make this easier, GemStone provides automatic transaction mode (autoBegin). In this mode, the session is always in transaction. Each commit or abort automatically starts a new transaction.

When you login through Topaz, your session is in automatic transaction mode. This allows you to make changes and commit them without having to explicitly start a transaction.

However, being in transaction on a system in which other users are also committing changes incurs a risk. GemStone must maintain your transactional snapshot view of the repository as long as you are in the transaction and do not commit or abort. These commit records require space in the repository, and in a rapidly changing system, an idle session in transaction can create a “commit record backlog”. Under worst-case conditions, this can cause the repository to fill up with this old data, run out of space, and shut down. If you are logging into a system that is in use, avoid remaining logged in and idle.

If you use topaz to log into a multiple-user system, you may wish to configure your system to default to not being in a transaction; this mode is manual transaction mode (manualBegin). In manual transaction mode, you are out of transaction until you explicitly begin a transaction (either by the **begin** command, or by executing a Smalltalk expression such as `System beginTransaction`), and when you commit, your session becomes out of transaction.

You can set topaz to be in manual transaction mode:

```
topaz> set transactionmode manualBegin
no transaction was in progress, transaction mode changed to
manualBegin
```

Note that if you are logged in, using this command will abort any transaction in progress, so it is better to execute this before login. You may wish to add this to the .topazini initialization file; see details starting on page 18.

You can also change the default for the entire system, by using the Stone configuration parameter STN_GEM_INITIAL_TRANSACTION_MODE. This sets the initial transaction mode for all sessions logging in, and may be particularly beneficial in production systems in which an inadvertent login can create problems.

See the *System Administration Guide for GemStone/S 64 Bit* for more information on managing transactions, handling sigAborts, and configuring STN_GEM_INITIAL_TRANSACTION_MODE.

WARNING
Idle topaz sessions should not be left in transaction in active multi-user systems. Even if the session is not in transaction and does not cause serious problems, your stale commit record requires extra work for the stone, and sessions may be terminated.

Other Types of Logins

X509-Secured

The GemStone server supports both traditional password-based login, and X509 secured certificate-based logins. These two types of login use a disjoint set of login parameters. When any of the traditional login parameters are set, all the X509 login parameters are unset, and vice versa. When you intend to authenticate using X509 certificates, do not set any of the following: **username**, **password**, **hostusername**, **hostpassword**, **gemstone**, **gemnetid**, or **sololigin**

For X509 login, the following are required: **cert, cacert, key, netldi**

and the following are optional: **logfile, directory, and extragemargs**

You must also create the appropriate keyfiles, and have a Stone that is running with a keyfile that allows X509 logins.

X509-Secured topaz and other logins are described in detail in the *X509-Secured GemStone System Administration Guide*.

Solo Logins

A solo login is a special kind of login that does not require a running Stone; it opens an extent file, and can execute GemStone Smalltalk code in the code environment of that extent. Since there is no Stone, a solo session is single-user, and changes to persistent objects cannot be committed. Solo sessions cannot, for example, run markForCollection in their own environments, nor execute methods that make or restore backups.

By default, a solo login uses the read-only extent in the GemStone distribution (`$GEMSTONE/bin/extent0.dbf`). However, you can configure it, by setting the Stone configuration parameter GEM_SOLO_EXTENT, to use another single extent file that contains scripting code or that contains your application code and data, provided that the following are true for the repository extent:

- The extent must not part of a multi-extent repository
- The extent file must either have read-only file permissions, or it will be exclusive-locked by the solo session.
- If not read-only, the extent repository must have been previously cleanly shutdown by the Stone

To login solo from topaz linked or RPC, execute `set solologin on`, then login.

For example:

```
topaz> set solologin on
topaz> login
[Info]: LNK client/gem GCI levels = 37000/37000
[Info]: Read-Only Repository:
/lark1/users/gsadmin/3.7/bin/extent0.dbf
[Info]: using libicu version 58.2
[08/08/2023 16:40:33.628 PDT]
  gci login: currSession 1  linked session
successful Solo login
topaz 1>
```

Any setting for **gemstone** is not used.

In topaz RPC, you may perform a solo login while also logged into a GemStone Stone, provided the extent file used by the Solo session is not in use by another solo session.

Object creation and memory use

Each Solo RPC or linked Gem also opens a 10MB read-write temporary file, `/tmp/gemRO_<pid>_extent1.dbf`, which is deleted on logout or process exit.

Object creation in a Solo session is limited to temporary object memory, but you may create objects as needed up to the limit of memory. To ensure there is sufficient memory, you may:

- Set a larger value for GEM_TEMPOBJ_CACHE_SIZE in the configuration file used by the topaz or Gem session.
- For linked sessions, use -T *cachesize* on the topaz command line.
- For RPC sessions, include -T *cachesize* in the NRS gemnetid login parameter. For example,

```
topaz> set gemnetid 'gemnetobject -T 200000'
```

Solo login with external connection to a running stone

While a solo session cannot run operations such as `markForCollection`, it can establish an external session to a running Stone and invoke operations such as `markForCollection` remotely.

See the *Programming Guide for GemStone/S 64 Bit* for details on GsTsExternalSession and GsExternalSession.

Multiple Execution Environments

The GemStone server has multiple compilation/execution environments available, which define separate environments for method lookup and code execution.

The default, and the location of the GemStone kernel, is environment 0. Most applications will only use environment 0 and can disregard references to execution environments.

The environmentId is set using commands:

`set compile_env:` *anInteger*

`env` *anInteger*

Most topaz commands operate on the current environment, and many accept an optional argument specifying the environmentId; see the command comments for exceptions.

1.2 Interacting with Topaz

Help Command

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

```
topaz 1> help time

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.

Topic?
```

You may enter further help topics at the `Topic>` prompt.

The enter key brings you back to the topaz command prompt.

For a full list of help topics, use **help** without specifying an argument.

Interrupting Topaz and GemStone

The Ctrl-C key combination interrupts Topaz and GemStone:

- 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.`

Logging Out

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

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

Logging out implicitly aborts your transaction.

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

These commands can include return values to the operating system shell; see **exit** (page 96) or **quit** (page 157) for details.

1.3 Executing GemStone Smalltalk Expressions

There are a number of commands allowing you to execute Smalltalk expressions: **run**, **printit**, **doit**, and **exec**. The following use of **printit**, for example, creates an instance of DateTime representing the current Date and Time:

```
topaz 1> printit
DateTime now
%
[41058561  DateTime]     a DateTime
  year                 2023
  dayOfYear            220
  milliseconds         75194099
  timeZone             [27058177  TimeZone]     a TimeZone
```

All of the lines after the **printit** command and before a 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.

The **doit**, **run**, and **printit** commands differ in the way the output is displayed; this is described in the next section.

The **exec** command allows the expression to be entirely on one line, which can improve readability of input and output files when there are many brief lines to be executed. **exec** can be used for multi-line commands: as for **run**, **printit**, and **doit**, a line beginning with % will indicate the end of the code to execute.

The printing is the same; you can simplify your output to not displaying the oops and classes of objects. To do this, use the topaz **omit oops** command.

For example:

```
topaz 1> omit oops
topaz 1> exec DateTime now %
a DateTime
  year                2023
  dayOfYear           220
  milliseconds        67995686
  timeZone            a TimeZone
```

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.

Strings vs. Unicode strings

GemStone Smalltalk supports two set of string classes; traditional Strings, including String, DoubleByteString and QuadByteString; and Unicode strings, including Unicode7, Unicode16, and Unicode32. Traditional Strings are native GemStone, while Unicode strings rely on the ICU libraries to provide sophisticated national language support. Both sets of string classes support the full Unicode character range. String classes are described in more detail in the *Programming Guide for GemStone/S 64 Bit*.

When you execute a Smalltalk expression in Topaz that creates a string object, the class of this object depends on the Topaz setting for SourceStringClass. The default is String, in which case a traditional string is created, of the lowest order in which all Characters of the result can be contained. Or it can be set to Unicode16, in which case a Unicode string is created, either a Unicode7, Unicode16, or Unicode32.

```
topaz 1> set sourcestringclass unicode16
topaz 1> display oops
topaz 1> exec 'hello' %
[4582145 sz:5 cls: 154369 Unicode7] hello
```

For further details, see **sourcestringclass** on page 174.

When a repository is in Unicode Comparison Mode, topaz detect this on login, and automatically changes the sourcestringclass to Unicode16, and the fileformat to utf8, and prints a message on this change on the topaz console. See the *Programming Guide for GemStone/S 64 Bit* for details on Unicode Comparison Mode.

Controlling the Display of Results

The result of the Smalltalk expression is returned to Topaz, and this object is displayed on the topaz console. Topaz provides several options to allow you to control the level of detail that is displayed for the object.

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 topaz command used to execute the code.

A level of 0 indicates that only the object itself is displayed; level 1 displays the object and its instance variables, level 2 include the instance variables of the instance variables of the object, and so on. The following examples are with **omit oops**.

The **printit** command always displays results with level 1:

```
topaz 1> printit
DateTime now
%
a DateTime
  year                  2023
  dayOfYear             220
  milliseconds          83048864
  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 **doit** command displays 0 levels:

```
topaz 1> doit
DateTime now
%
08/08/2023 15:04:35
```

The **run** command uses the current level setting to display the results. By default, this is 0, and produces the same display as the **doit** command. You can set the level explicitly using the **level** command, to display more of the object.

For example, at the default level 0, the **run** command produces the same display as the **doit** command:

```
topaz 1> level 0
topaz 1> run
DateTime now
%
08/08/2023 15:05:23
```

Setting the level to 2 would give this view:

```
topaz 1> level 2
topaz 1> run
DateTime now
%
a DateTime
  year                  2023
  dayOfYear             220
  milliseconds          83121913
  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 for the objects that are within each of DateTime's instance variables. 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

By default, Topaz attempts to display all of a result, no matter how long. So for example if an expression returns a collection, every item in the collection will be displayed.

The **limit oops** controls how much Topaz displays of pointer or NSC objects, such as Arrays and Sets, that come back as a result. Similarly, **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. To avoid

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
...(9 more bytes)
```

Setting the limit to 0 restores the default condition.

Displaying Variable Names, OOPs, and Byte Values

Two complementary commands, **display** and **omit**, control some features of how objects are displayed in topaz. For more details on the options, see **display** (page 84).

OOP Values

It's useful in debugging to be able 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, except for certain special objects, including SmallIntegers, Characters, Booleans, and nil. For example, displaying the objects

```
topaz 1> level 1
topaz 1> display oops
topaz 1> exec { 123 . 3.4 . #abc . Bag with: 'x' } %
[41006593 size:4  Array] a Array
  #1 123
  #2 [9273812352681325366  SmallDouble]  3.4
  #3 [15625729 size:3  Symbol] abc
  #4 [41005569 size:1  Bag]   a Bag
```

Objects that have headers include the object's OOP (a 64-bit unsigned integer), the size (for indexable or NSC objects), and the object's class name.

All OOP Values

Using the **display alloops** command, you can also display headers for specials, and additional information.

```
topaz 1> level 1
topaz 1> display alloops
topaz 1> exec { 123 . 3.4 . #abc . Bag with: 'x' } %
[41008641 size:4 primitiveSize:4 cls: 66817 Array] a Array
  #1 [986 size:0 primitiveSize:0 cls: 74241 SmallInteger] 123 == 0x7b
  #2 [9273812352681325366 size:0 primitiveSize:0 cls: 121345 
SmallDouble]  3.4
  #3 [15625729 size:3 primitiveSize:3 cls: 110849 Symbol] abc
  #4 [41007617 size:1 primitiveSize:6 cls: 102657 Bag] a Bag
```

You can turn off the display of all or some OOPs by typing **omit oops** or **omit alloops**, at the Topaz prompt.

Byte Values

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

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

Entering **omit bytes** or **omit decimalbytes** restores the default display mode.

Committing and Aborting Transactions

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.

The topaz command **begin**, or executing `System beginTransaction`, may also be used to begin a transaction if you are not in automatic transaction mode, or to abort your transaction.

Although you can abbreviate most Topaz commands and parameter names, **commit**, **abort**, and **begin (**as well as some others such as **quit** and **exit**) must be typed in full.

Importing files: topaz commands and GemStone code

The Topaz interactive environment takes its input from standard input, such as your terminal. You can also use the **input** command to make Topaz take its input from a file. Topaz treats the lines in the input file as if they were entered on the command line; this file may contain commands to perform work, or code to be filed in; there is no difference.

For example, the following command, 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.

When you have code that was filed out of GemStone, the input command allows you to load this code into another Stone. You will need to check for errors and commit to make the code persistent.

```
topaz 1> input $HOME/animal.gs
<input by default echoes all input>
topaz 1 +> errorcount
0
topaz 1 +> commit
Successful commit
```

Input files can be nested. The prompt adds a + to indicate the depth of nesting of input files.

Batch processing goes very quickly. It is a good idea to record your topaz session to a file, use **output push**, so you can check for errors. This is described starting on page 31.

Handling text outside the ASCII range

Character with codepoints between 128 and 255 are outside the ASCII range, but the codepoints only require one byte. Since UTF-8 encoding is identical only for characters with up to 7 bits, how Characters in this range are encoded in a text file or paste buffer depends on how your system is configured.

If all your code and text file contents, and all commands entered via scripts, are limited to the ASCII range, there is no difference in behavior and your system requires no further configuration.

Terminal input such as pasting into your command line is always decoded from UTF-8. If you are copying from a text editor, and the text you are pasting includes characters outside the range, ensure that the copy buffer encoding is UTF-8.

Text files that are input into Topaz are interpreted based on Topaz's **fileformat** setting. The default is UTF8; text files (that do not include a fileformat command in the header) are assumed to be encoded as UTF-8, and it will attempt to decode any bytes in the 128-255 range.

To input text files that are encoded as 8BIT into Topaz, use:

```
topaz> fileformat 8BIT
```

If you use any characters with codepoints over 127, the fileformat must be set according to the encoding of the input file, otherwise the results will be corrupted.

The same considerations apply for text files output from topaz. Log files produced by the **output** command are always encoded as UTF-8. Code fileouts produced by the topaz **fileout** command include a header line specifying the file format, so these files can be filed in again without problems.

Topaz displays characters in the range 0-31 using caret notation, for example, Character cr is displayed as ^M. The exceptions are Character lf and Character tab, which control display as designed. Characters with codePoints 128-159 and Characters with codePoints over 255 are displayed using C/Java hex format, for example, \u012c.

Capturing Your Topaz Session In a File

It's often useful to keep a record of your interactions with GemStone during testing and debugging, and when executing scripts to simplifying finding errors.

You can do this with the Topaz command **output**. 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).

For example, either of these lines cause all subsequent interactions to be captured in a file called `animaltest.log`:

```
topaz 1> output animaltest.log

topaz 1> output push animaltest.log
```

Using **output** alone or **output push**, if the file you name doesn't exist, Topaz creates it. Under UNIX, if you name an existing file, Topaz will overwrite the previous file.

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

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

To have topaz create a new file if a file with the given name already exists, use **pushnew**:

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

If `animaltest.log` already exists, the file `animaltest_1.log` is created.

To stop writing to the current log, use **output pop**.

```
topaz 1> output pop
```

Writing to multiple log files

As the command names **push** and **pop** imply, Topaz can maintain a stack of up to 20 output files. Topaz input and results are written to each file on the stack.

You can specify topaz only write only to the file on the top of the stack using the keyword **only**.

For example:

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

The keyword **only** will also prevent the results of any topaz commands from being displayed on your screen; they will only be written to the log file.

The following sequence would capture one set of commands in the file `mathtest.log`, and a second set of commands in `mathtest2.log`:

```
topaz 1> ! Capture the next command and result in mathtest.log
topaz 1> output push mathtest.log
topaz 1> time
08/08/2023 14:49:47.558 PDT
CPU time:   0.013 seconds
topaz 1> exec 5 * 8 %
40
topaz 1> ! Capture the next command and result in mathtest2.log
topaz 1> output push mathtest2.log only
topaz 1> time
topaz 1> exec 5 * 9 %
topaz 1> ! Close mathtest2.log and resume using mathtest.log
topaz 1> output pop
```

Notice that the result of the second time and the result of 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 the exclamation mark **!** in this example, to indicate a comment line. You can use either exclamation point or pound sign as the first character in a line, or the command **remark,** to begin a comment. Comments are important for annotating Topaz input files created for batch processing or testing.

Invoking Operating System Functionality from Topaz

From within topaz, you can easily execute operating system commands, or escape to an operating system shell and execute commands directly on the command line. To do this, invoke the topaz **shell** command.

You can enter your operating system command on the **shell** command line, as for example:

```
topaz> shell echo $GEMSTONE
/lark1/users/gsadmin/GemStone64Bit3.7.0-x86_64.Linux

topaz>
```

Or you can enter **shell** with no arguments, in which case you can interactively enter a sequence of operating system commands. When you are done, type **`exit`** or control-D to returns to topaz.

```
topaz> shell
% echo $GEMSTONE
/lark1/users/gsadmin/GemStone64Bit3.7.0-x86_64.Linux
% exit

topaz>
```

You can, of course, also execute operating system functionality from GemStone Smalltalk using:

```
topaz 1> run
System performOnServer: codeToBeExecuted
%
```

1.4 Using Topaz for Code Development

Topaz, in conjunction with methods in GemStone Smalltalk, can be used to examine and write classes and methods.

Creating 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        a Array
  constraints         nil
  classVars           nil
  methDicts           nil
  poolDictionaries nil
  categorys           nil
  primaryCopy         nil
  name                Animal
  classHistory        a ClassHistory
  transientMethDicts a Array
  destClass           nil
  timeStamp           a DateTime
  userId              DataCurator
  extraDict           nil
  classCategory       nil
  subclasses          nil
```

Once you have a class, you can create methods for it in topaz.

You can begin the definition of an instance method by issuing the **method:** command at the Topaz prompt. This command takes a single argument: the name of the class for which the method will be compiled.

```
topaz 1> set category 'Accessing'
topaz 1> method: Animal
habitat
      "Return the value of the instance variable 'habitat'."
      ^habitat
%
```

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

```
topaz 1> set category 'Instance creation'
topaz 1> classmethod: Animal
returnAString
      "Returns an empty String"
      ^String new
%
```

Topaz maintains state for the Class and method category, so if you have multiple methods, you do not need to specify these values for each method definition. Expressions such as the above will set the Class and the category, and this will be used for subsequent methods until changed.

To set the current class, use the **set class** command:

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

To set the current category, use the **set category** command:

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

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

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
```

Once you've established a class and a category, you can define a method simply:

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

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.

Using a Text Editor to Edit Methods

The **edit** command allows you to use a host text editor to edit GemStone Smalltalk expressions, including method text.

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 command that defined Smalltalk expressions: including printit, exec, run, doit, method, and classmethod. When you exit the editor, Topaz saves the edited text in a temporary file and asks you whether you'd like to compile the altered code. If you type **y** or **yes**, Topaz effectively reissues your command with the new text.

For example, to edit the existing instance method `habitat:`, you would enter **edit** as shown below:

```
topaz 1> edit Animal >> habitat:
```

To edit an existing class method, use an expression of the form:

```
topaz 1> edit Animal class >> returnAString
```

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

Listing Methods, Categories, and other information

The topaz list command provides a way to find out information about classes and methods, as well as other information about your system.

To list the instance method selectors for a class, use **list selectors**:

```
topaz 1> list selectors
  habitat
  habitat:
```

You can filter the list to limit on the initial characters of a selector (case insensitive).

To see only the class methods in System that that with 'gemenv':

```
topaz 1> set class System
topaz 1> list cselectors gemenv
  gemEnvironmentVariable:
  gemEnvironmentVariable:put:
```

To see the categories and methods that are in the current class, use **list categoriesin**. Topaz lists all of the class and instance method selectors in the selected class by category.

```
topaz 1> list categoriesin
----------------- Instance Methods:
category (as yet unclassified)
  habitat
category Updating
  habitat:
----------------- Class Methods:
category (as yet unclassified)
  returnAString
```

To list the source code of an instance method, use **list method:**

```
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.

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. The list command by itself is used in debugging, to display source for a method on the current execution stack. For more information, see the options described under **list** starting on page 125.

Filing Out Classes and Methods

You will commonly want to file out GemStone Smalltalk code at some level of granularity, by Method, Method Category, Class, or SymbolDictionary. Some reasons to do this are:

- 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.

GemStone fileouts are in topaz executable form. You can process this text using editors or other operating system utilities and then execute it with the Topaz **input** command.

Fileouts are commonly done using GemStone Smalltalk methods, including *Class* `fileOutMethod:on:`, *Class* `fileOutCategory:on:`, *Class* `fileOutClassOn:`, and similar methods; and `ClassOrganizer fileOutClassesAndMethodsInDictionary:on:`. For more information on these methods, see the methods in the image.

Fileout can also be performed using the Topaz **fileout** command. For example, 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:

```
fileformat utf8
set sourcestringclass String
set compile_env: 0
! ------------------- Class definition for String
expectvalue /Class
doit
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 other subcommands, to allow you to fileout a method or other granularity of code. For more details, see the options under **fileout** starting on page 105.

Code outside the ASCII range

The **fileout** command encodes the resulting file according to the current **fileformat** setting. By default, this is utf8.

To configure your system to fileout in 8-bit extended ASCII, for compatibility with older code, you may specify this using:

```
topaz 1> fileformat 8BIT
```

The current fileformat and SourceStringClass settings are automatically added to the results of **fileout**. This ensures that the same format is used to filein as was used to fileout. Do not removes these statements, unless you are certain the correct settings will be used on any later filein; the incorrect settings can cause the input to be corrupted.

1.5 Advanced Topaz features

The features previously described to manage your session, execute Smalltalk server code, and accept input from and write to files, are sufficient for most application needs.

Topaz also provides additional features that support sophisticated debugging of GemStone Smalltalk code and streamline access to specific objects.

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.

```
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. E.g., if an indexed object also had three named instvars, 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 ways you can specify objects in Topaz commands.

Object Specification Formats

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

integer
An Integer.

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*]]

`$`*character*
A literal Character.

aVariableName
This can be either a GemStone Smalltalk variable name (such as a Class name, or a variable set in a SymbolDictionary), a local variable created with the **defin.e** command, or a predefined Topaz variable. See "Topaz Variables" on page 39

`**` 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).

`'`*aLiteralString*`'`
A literal String.

`#`*aLiteralSynbol*
A literal Symbol (no white space allowed).

`#'`*a Quoted Symbol*`'`
A quoted literal Symbol.

Example 1.1 Using OOP and * (last result) specifications

```
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
$n
```

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, which is very unlikely to be `@1335297`.

Topaz Variables

Topaz lets you refer to any object in a command by using the OOP of that object. Long numerical OOPs are difficult to work with, so Topaz also allows you to define local Topaz variables to refer to the OOPs by name. This is done using the **define** command.

Since Topaz is flexible about resolving strings to names, use some care in choosing the names of variables. Names that are set using **define** take precedence over Smalltalk variable names (such as the names of Classes), and over the Topaz build-in variables (see the next section).

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. Local variables are case-sensitive.

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.

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
------------------------
      CurrentMethod       = nil
      ErrorCount          = 2
      CurrentCategory     = nil
      CurrentClass        = nil
      LastResult          = nil
      LastText            = nil
      myUserProfile       = 1458177
```

define reports values as OOPs rather than literals; so an ErrorCount of 2 means that there were zero errors.

In this status report there are two sections. User-defined local variables are listed first, followed by the topaz predefined variables. These are local variables that Topaz automatically creates for you:

CurrentMethod—the OOP of the current method (set by lookup or list)

ErrorCount—the OOP of the count of Topaz and GemStone errors since Topaz started.

CurrentCategory—the OOP of the current category

CurrentClass—the OOP of the current class

LastResult—the OOP of the last execution result

LastText—the OOP of the text of the last GemStone Smalltalk expression executed or compiled

myUserProfile—the OOP of UserProfile for the current session's login.

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.
```

Sending Messages

In addition to the usual ways of sending messages with Smalltalk code, Topaz allows you to send messages to an object identified by any of the means described in "Specifying Objects" on page 38. This lets you use OOPs or Topaz variables directly.

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.
- Each item must be delimited by one or more spaces or tabs.

Chapter 2 Scripting with Topaz and SuperDoit

Many tasks in any software environment can be made easier or automated using scripts. GemStone is no exception, and operations such as markForCollection and backup are often scripted to start automatically at certain times.

In GemStone, scripts rely on **topaz**, GemStone's command line interface. Topaz itself does not provide flow of control, nor conditional execution. For fully featured scripting, you can combine topaz with shell scripting such as bash, or use the SuperDoit environment, to allow script arguments, conditional execution, create sequences of operations, and similar scripting requirements.

Standard and Solo logins

A standard GemStone login requires a running Stone, and creates a session in a GemStone multi-user transactional environment. The login can perform any operation that its GemStone and UNIX UserId allows, including markForCollection and other operations that are often scripted.

There are tasks, however, that do not actually require a Stone to be running, and can be done with less risk and lower impact by using solo logins. Solo logins, described under "Solo Logins" on page 22, allow a single-user, read-only login that can load and execute GemStone Smalltalk code.

With a solo login:

- You can execute Smalltalk code, without impacting other users.
- Using an empty extent and a non-privileged user, anyone can execute GemStone Smalltalk code, such as for text processing, with minimal security concerns.
- A solo session can filein topaz files containing GemStone classes and methods and execute code invoking them, provided that code execution does not require a Stone, nor perform a commit or abort. Code that attempts to commit or abort will signal an error.
- You **cannot** perform operations that require a Stone and are inherently multi-user, such as markForCollection or backup.

- You can use GemStone’s external session protocols to login remotely to a running Stone to perform operations such as markForCollection or backup. This requires the login credentials for the running Stone.

While solo logins cannot replace standard logins for the most common scripting needs, they provide additional options for executing GemStone code from the command line.

2.1 Topaz scripting

Creating a script

To execute markForCollection or other simple commands, only a simple script is needed.

1. Create the topaz script

To start with, create a file containing the login instructions and the GemStone code you wish to execute.

For example, this might be the contents of a text file script that runs markForCollection, with the name `runmfc.topaz`:

```
set user DataCurator password swordfish gemstone gs64stone
login
! run garbage collection mark/sweep
exec SystemRepository markForCollection %
exit
```

2. Verify using the topaz input command

The topaz input command can execute any file containing topaz commands, such as the `runmfc.topaz` just created. Since the login information is embedded in the contents of this file, you do not need to do a separate login.

To verify your script, start topaz and input the script file. Note that if successful, this will in fact run markForCollection on your Stone.

```
topaz 1> input runmfc.topaz
```

3. Pass the script on the topaz command line.

The topaz executable is invoked on the command line, and for scripting you can pass in a the topaz script to be executed on the command line, using the **-I** or **-S** topaz argument.

- The **-I** command, (initialization) suppresses automatic use of a `.topazini`, and suppresses echo of output to the console.
- The **-S** command (scripting) does read a `.topazini`; this can be suppressed by also specifying **-i**. The **-S** also does not suppress echoing output; this can be suppressed manually by also specifying **-q**. In other words, **topaz -I** *scriptName* is equivalent to **topaz -i -q -S** *scriptName.*
- You may pass in two scripts, using both **-S** and **-I** scripts; the **-I** script is always executed first.

For example:

```
unix> topaz -I runmfc.topaz > MFC.out
```

4. Further embed in a UNIX shell script

To allow a simple shell command to execute the code, rather than requiring the topaz arguments and redirect, you can embed the topaz script within a UNIX shell script.

For example, create a file `runMFC` with the following contents:

```
#! /bin/bash
#set -x
$GEMSTONE/bin/topaz -l -I runmfc.topaz <<EOF >>MFC.out
EOF
```

Alternatively, you can include the topaz script code in the shell script. For example:

```
#! /bin/bash
#set -x
$GEMSTONE/bin/topaz -il <<EOF >>MFC.out
set user DataCurator password swordfish gemstone gs64stone
login
! run garbage collection mark/sweep
exec SystemRepository markForCollection %
exit
EOF
```

Either script can be executed (once given the appropriate permissions) by executing:

```
unix > ./runMFC
```

Security Concerns

The above scripts embed the GemStone password in the script, which may not be appropriate, depending on the security requirements for your environment.

You can segregate the login details into a separate file, which can be automatically located following the rules for `.topazini`, or by passing in using the **-I** topaz argument.

Using a separate initialization file allows you to move the user name and password to a separate file, and thus not visible in the script code itself.

Given the following two scripts:

`runmfc.topaz`:

```
set gemstone gs64stone
login
! run garbage collection mark/sweep
exec SystemRepository markForCollection %
exit
```

`myini`:

```
set user DataCurator password swordfish
```

Then you can execute, for example:

`unix>` **topaz -I myini -S runmfc.topaz > MFC.out**

Or skip creating the file `runmfc.topaz`, and create a UNIX script with the following contents:

```
#! /bin/bash
#set -x
$GEMSTONE/bin/topaz -l -I myini <<EOF >>MFC.out
set gemstone gs64stone
login
! run garbage collection mark/sweep
exec SystemRepository markForCollection %
exit
EOF
```

2.2 Scripting with topaz she-bang

When writing solo scripts (that do not require a Stone), you can avoid writing shell script code altogether. Topaz she-bang scripting has some restrictions:

- The environment variable $GEMSTONE must be set
- she-bang scripts execute in a solo session, that is, do not log into a running stone.
- she-bang scripts run in a linked topaz session
- no topaz command line options can be passed in
- No `.topazini` files are read

The first line of the she-bang can be defined two ways:

`#!/usr/bin/env topaz`
: `$GEMSTONE/bin` must be on the machine executable search path.

`#!`*fullPathToExecutable*`/topaz`
: `GEMSTONE/bin` does not need to be on the machine executable search path, but *fullPathToExecutable* must be the full path, not including environment variables.

For example, if you define a executable text file, `myscript`, with the following contents:

```
#!/usr/bin/env topaz
set user DataCurator password swordfish
login
run
  | files sz |
  files := GsFile
      contentsOfDirectory: '$GEMSTONE/data/tranlog*.dbf'
      onClient: false.
  sz := 0.
  files do: [:ea |
      sz := sz + (GsFile sizeOfOnServer: ea)
      ].
  'tranlogs consume ', (sz / 1024) asInteger asString, ' KB'.
%
```

This file can be executed, with or without a running stone, to report the sum total size of tranlog files in the given directory.

Note that this will (in a default configuration) execute using the empty distribution extent, and checks the file sizes on disk rather than using any internal representation in the repository that is actually generating the transaction logs.

Topaz exits when the script completes.

2.3 Script Arguments

The Topaz command line supports additional arguments, which can appear after a double dash at the end of the command line. Anything after this double dash are ignore by topaz.

These command line arguments are available from Smalltalk code using `System commandLineArguments`.

```
unix> topaz -l -- these are some arguments
...
topaz 1> exec System commandLineArguments printString%
anArray( 'topaz', '-l', '--', 'these', 'are', 'some',
'arguments')
```

Passing arguments using bash scripts

By using bash to invoke topaz, you can create executables with command line arguments, that use topaz scripting to perform operations.

The following example uses three files to demonstrate writing a command-line script with one argument, that executes GemStone Smalltalk code.

The top-level bash script invokes topaz with the login information in a topaz initialization file and the actual script file, passing in a single argument. For simplicity, this example does no error checking. An actual script, of course, should check that the arguments are present and valid.

Example 2.1 Example files demonstrating bash script with argument

Contents of executable gettranlogspace:

```
#!/bin/bash
export GEMSTONE=/lark1/users/gsadmin/3.7
$GEMSTONE/bin/topaz -lq -I $GEMSTONE/scripts/myini -S
   $GEMSTONE/scripts/fileSpaceUsed.tpz -- $1
```

This invokes linked topaz (**-l**), passing in the initialization file (**-I**) and the script file (**-S**). The use of the **-q** flag suppresses topaz output that will clutter the output. Arguments following **--** are passed to topaz, in this case, the bash shell's (0-based) second element, which is the argument (a file system path).

By setting $GEMSTONE explicitly and specifying the path to topaz, there is no need to perform GemStone environment setup before executing this script; it can be run in any environment.

Contents of file myini:

```
set user DataCurator pass swordfish
set gemstone gs64stone
set solologin on
```

This sets solo login, so no stone login is required. This script logs in as DataCurator, but another userId with limited privileges would be sufficient, since the script will only perform limited operations.

Contents of file fileSpaceUsed.tpz:

```
login
run
  | dir files sz |
  dir := System commandLineArguments last.
  files := GsFile
        contentsOfDirectory: dir
        onClient: false.
  sz := 0.
  files do: [:ea | sz := sz + (GsFile sizeOfOnServer: ea)].
  GsFile gciLogClient: dir, ': files consume total ',
   (sz / 1024) asInteger asString, ' KB'.
%
logout
exit
```

The use of `System commandLineArguments` gets the full set of tokens that invoked topaz, so the last element is the bash argument. Again, for simplicity no error checking is done in this example.

Executing the example:

The script executes and reports the amount of space used by all files in the given directory. This is not necessarily

```
unix> ./gettranlogspace /lark1/users/gsadmin/tranlogs
/lark1/users/gsadmin/tranlogs: files consume total 98477 KB
```

2.4 SuperDoit

The SuperDoit scripting environment allows you to write command line scripts entirely in Smalltalk. SuperDoit uses the topaz features described earlier in this chapter, and provides a framework for handling arguments and usage sections, as well as other features that support writing complex scripts.

To execute SuperDoit scripts:

- $GEMSTONE must be defined
- $GEMSTONE/bin must be on your search path
- For stone scripts, $GEMSTONE/version.txt must be present.

Script Types

SuperDoit scripts come in two variants:

- **stone**
  executes the script as a normal login to a running Stone. Login details are provided in a topaz initialization file (`.topazini`, or another name), which may be provided on the command line using the **-I** argument, or in one of the standard `.topazini` locations: the current working directory, or the home directory of the user executing the script.

  All the details required for a login to a Stone in your specific environment must be available, in the topaz initialization file, $GEMSTONE_NRS_ALL, and/or other environment variables.

  By convention, these scripts end in `.stone`. You may use the template `$GEMSTONE/examples/superDoit/template.stone` to start developing your script.

- **solo**
  executes the script as a solo session. Commits, and operations that require a stone, are not allowed, and solo scripts cannot login RPC. Solo scripts may use the rowan extent, `$GEMSTONE/bin/extent0.rowan.dbf` (as was the default in previous releases) for their solo extent, or the base GemStone image, `$GEMSTONE/bin/extent0.dbf`. The only time the distinction would matters if you are using or avoiding Rowan-specific behavior.

  Solo scripts do not automatically look for a standard `.topazini` file. Solo scripts use a solo-specific topaz initialization file, located within the SuperDoit project, which logs in as SystemUser. You may use the **-I** argument to specify a different topaz initialization file.

  By convention, solo scripts end in `.solo`. For clarity, to distinguish scripts that specifically use the base Gemstone image or the Rowan extent, the script may end in `.gs_solo` or `.rw_solo`. You may use the template `$GEMSTONE/examples/superDoit/template.rw_solo` or `$GEMSTONE/examples/superDoit/template.gs_solo` to start developing your script.

Script structure

A SuperDoit script consists of a number of sections, each one started by a keyword, and ending with %. All sections are optional.

The Smalltalk code that the script executed is in a section with the keyword **doit**. Note that while this looks similar to topaz syntax, it is a keyword defining the code execution section; topaz syntax is not understood in solo and stone scripts.

A minimal SuperDoit script might be, for example,

```
#!/usr/bin/env superdoit_solo
doit
   Time now asString
%
```

In addition to a doit section, there are number of sections that you may include depending on your script’s requirements. Most commonly, you will want to include sections for script command line options, in the options section, and usage information in a usage section.

An example of a script with the options, usage, and doit sections:

```
#!/usr/bin/env superdoit_stone
# Print the contents of DbfHistory with a byte limit
options
   {SuperDoitOptionalOptionWithRequiredArg
     long: 'bytes' short: 'b'}
%
usage
   $basename [-h] [-D] [-b <number>] [-- [-I <topazini>]]
   Return DbfHistory for the stone specified by <topazini>, or
   if not specified, the default .topazini for the environment.
   The optional -b argument specifies to return only the first
   <number> bytes of the DbfHistory.
%
doit
   self bytes isNil
      ifTrue: [DbfHistory]
      ifFalse: [DbfHistory copyFrom: 1 to:
         (self bytes asInteger min: DbfHistory size)]
%
```

This script show how access to the argument is handled: `self bytes` returns the contents of the command line argument specified using `-b` or `--bytes`.

Also note that the usage here is simplified; the arguments `--help`, `--Debug`, and `--bytes` may also be used on the command line. See the section under “Arguments” on page 51.

Finally, the argument to topazini that is specifically mentioned in the usage is only one of a number of arguments to topaz that can be passed to the SuperDoit script and forwarded to topaz. These arguments, if used, are separated from the script arguments by --. See the section “Arguments to Topaz” on page 53.

Commonly Used Sections

doit
holds the body of the script

options
specifies command line arguments; see the possible specifications under “Arguments” on page 51.

usage
provide usage text. This can be in any format; all text between the usage and closing % is printed in response to the -h or --help argument to the script.

input
specifies code (topaz or gs) that is filed in prior to executing the doit.

customoptions
Allows you to override the default arguments for help and debug.

Complex Scripting

The following sections allow functionality to be broken down into supporting methods and associated state.

method
Define a method on the script itself

instvars
Define instance variables for the script, which can be accessed by the methods defined in method, method: and classmethod: sections.

method:
Define an instance method on an existing class.

classmethod:
Define a class method on an existing class.

Rowan Support

The following are advanced sections to support Rowan repositories:

projectshome
Declare the value of the ROWAN_PROJECTS_HOME environment variable.

specs
Specify an array of Rowan load specification STON objects used to load external projects into the image.

specurls
Specify a list of spec urls that reference the location of a Rowan load specification STON object.

Script Comments

You may add comments to a superDoit script by prefixing the line with #.

These comments can be put in anywhere within the script, other than within Smalltalk code, where comments must use regular Smalltalk syntax (double quotes).

Arguments

Command line arguments may be defined using name-value pairs, names without values, or as positional arguments.

The options section defines the named and name-value command line arguments; positional arguments are accessed in the doit section, sending `self` `positionalArgument` to return the array of zero or more positional arguments that were included on the command line.

Named and name-value pairs can be specified either using getopts syntax `-`*Character value* (referred to as "short" in SuperDoit), or `--`*String=value* ("long").

The long version is always supported; you may also specify a short version when defining the script arguments.

Arguments with values may be required or optional, and optional arguments may have a default value specified. See the specifications on "Defining Script Arguments" on page 52.

Standard arguments

By default, all scripts have the help and debug options:

-h or --help
: Returns usage information specified in the usage section

-D or --Debug
: If an exception occurs, the script stops in topaz rather than exiting. This allows you to debug using topaz commands such as where.

--gemDebug
: If terminal is connected to stdout, bring up debugger. If not, dump stack to stdout and wait for topaz to attach using topaz DEBUGGEM command.

To exclude help and/or debug options, you can define a section in your script for customoptions.

Note that since the help section is processed and returned in Smalltalk code, executing a script to return the help text requires a successful login to the Stone or solo extent.

Defining Script Arguments

The script options are specified as an array of objects (in GemStone Array Builder syntax, curly-brace delimited and period separated), specified using the following expressions.

```
SuperDoitOptionalOptionWithNoArg long:
SuperDoitOptionalOptionWithNoArg long:short:
SuperDoitOptionalOptionWithRequiredArg long:
SuperDoitOptionalOptionWithRequiredArg long:default:
SuperDoitOptionalOptionWithRequiredArg long:short:
SuperDoitOptionalOptionWithRequiredArg long:short:default:
SuperDoitRequiredOptionWithRequiredArg long:
SuperDoitRequiredOptionWithRequiredArg long:short:
```

For example, to specify an argument that may be supplied with `-b` *value*, with `--bytes` *value*, or omitted:

```
options
   {SuperDoitOptionalOptionWithRequiredArg
     long: 'bytes' short: 'b'}
%
```

Arguments to Topaz

In addition to your script-specific arguments, you may also include arguments that are directives to topaz. These are included following a -- on the command line.

For example, to pass in both a script argument and a topazini:

```
getDbfHistory.stone -b 100 -- -I .topazini
```

This can be used to pass in any other topaz argument, for example to override the default use the rowan extent to execute a SuperDoit script:

```
simple.solo -- -C GEM_SOLO_EXTENT=$GEMSTONE/bin/extent0.dbf
```

Most topaz directives can be used, and will override those specified by the SuperDoit infrastructure or the configured environment. Any use of the topaz -S option is ignored, and solo scripts may only log in linked.

Usage

The usage section includes a section of text that will be displayed when the script is invoked with **-h** or **--help**.

```
usage
   $basename [-h] [-D] [-b <number>] [-- [-I <topazini>]]
   Return DbfHistory for the stone specified by <topazini>, or
   if not specified, the default .topazini for the environment.
   The optional -b argument specifies to return only the first
   <number> bytes of the DbfHistory.
%
```

The usage section is optional. If it is not included in the script, when **-h** or **-help** is invoked, the default help with **-h/--help/-D/--debug** options is displayed.

Coding in the doit section

The doit section consists of GemStone Smalltalk code. Within the doit, self corresponds to the instance of SuperDoitExecution. This provides access to the script arguments and environment, and set of convenience helper methods.

Available methods include (but are not limited to):

`executionStoneName`
: Returns the name of the running stone; for solo, this will be gs64stone.

`globalNamed:` *globalName*
: Return a global variable with the given name.

`globalNamed:` *globalName* `ifAbsent:` *notfoundblock*
: Return a global variable with the given name, executing the *notfoundblock* if it does not exist.

`positionalArgs`
: Returns an array containing all positional arguments from the command line.

`stderr`
: The stderr that the script writes to.

`stdout`
: The stdout that the script writes to.

`noResult`
At the end of the script, this allows you to return with no messages on the command line.

`exit:` *errmsg* `withStatus:` *anInteger*
Exit the script, writing *errmsg* to stderr and with the OS process exit status set to *anInteger.*

`exitWithStatus:` *anInteger*
Exits the script, with the OS process exit status set to *anInteger.*

`isSolo`
Return true if in a solo session.

`log:` anObject
Write the argument in STON format to stdout.

`logErrorMessage:` *aString*
Write the given string on stderr.

`logMessage:` *aString*
Write the given string on stdout.

`commandLine`
Returns the command line for the script.

Output

At the end of a superDoit script execution, it displays the final object (or the exception, if an exception occurred), as a printable String, or the exception if an error occurred.

To avoid this output, the final statements of the script can send `self noResult`.

Examples and Templates

$GEMSTONE/examples/superDoit/

This directory contains a number of example scripts, which can be executed as is, and script templates, illustrating the use of arguments and error handling.

Rowan-specific GsCommands examples

The scripts in `$GEMSTONE/examples/GsCommands/` show the use of the Rowan GsCommand package with SuperDoit scripts, illustrating the use of the Rowan-specific sections supported by SuperDoit scripts.

Chapter 3 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 4.

3.1 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, although not all step points are legal for breakpoints.

Generally, step points correspond to the message selector and, within the method, message-sends, assignments, and returns of nonatomic objects. Compiler optimizations, however, 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:

Example 3.1 listing step points in a method

```
topaz 1> set class Dictionary
topaz 1> list steps method: removeKey:ifAbsent:
   removeKey: aKey ifAbsent: aBlock
 * ^^                              1,2                               *******
   "Removes the Association with key equal to aKey from the receiver and returns
   the value of that Association.  If no Association is present with key
   equal to aKey, evaluates the zero-argument block aBlock and returns the
   result of that evaluation."

   | anAssoc |
   anAssoc:= self removeKey: aKey otherwise: nil .
 *        ^4      ^3                                                 *******
   anAssoc == nil ifTrue:[
 *         ^5                                                        *******
     aBlock == nil ifTrue:[^ nil ].
 *          ^6              ^7                                       *******
     ^aBlock value
 *   ^9      ^8                                                      *******
     ].
   ^ anAssoc value
 * ^11      ^10                                                      *******
```

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 to step through this method, the first step stops execution at the beginning of the method. The next step takes you to the point where `removeKey:otherwise:` is about to be sent to `self`. Stepping again would execute that message-send and halt execution at the point where `anAssoc` is about to be assigned. Another step would cause that assignment to be happen, and then halt execution just before the message == is sent to `anAssoc`.

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).

You can use the **break** command to set a method breakpoint at a particular step point within a method.

While you can set a breakpoint on any method, methods with optimized selectors, 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. Note that in the step points listed above for `Dictionary >> removeKey:ifAbsent:`, above, there are no step points on `ifTrue:`.

Breakpoints

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

`break` *aClassName* `>>` *aSelector* `[@` *stepNumber*`]`
`break` *aClassName* `class >>` *aSelector* `[@` *stepNumber*`]`

For example:

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

Establishes a breakpoint at step point 2 of the class method `openRead:` for GsFile. There are a number of ways to specify the specific class and method: see **break** on page 71 for details.

The **break list** command allows you to see all breakpoints set

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

In the **break list** result, 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
   topaz 1> break list
   1: GsFile >> nextLine @ 1
   2: GsFile class >> openRead: @ -2 (disabled)
   3: String >> < @ 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
```

3.2 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 Animal >> name:
```

Then run this code:

```
topaz 1 > run
Animal new name: 'Dog'
%
a Breakpoint occurred (error 6005), Method breakpoint encountered.
1 1 Animal >> name:                              @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:                        @1 line 1 [methId 25534209]
2 Executed Code                              @3 line 1   [methId 25504513]
3 GsNMethod class >> _gsReturnToC              @1 line 11 [methId 4912641]
  [GsProcess 27551489]
```

The **frame** command allows you to see more information about the instance and temporary variable names for a particular stack frame. With level 0 (the default), only the variable values themselves are displayed; increasing the level displays durther details.

For example, at the defaul level 0:

```
topaz 1> frame 1
1 Animal >> name:                                  @1 line 1
    receiver [45404929  Animal]       anAnimal
    newValue [45404161 size:3  String] Dog
```

and with level 1:

```
topaz 1> level 1
topaz 1> frame 1
1 Animal >> name:                                  @1 line 1
    receiver [45404929  Animal]       a Animal
  name                nil
  favoriteFood        nil
  habitat             nil
    newValue [45404161 size:3  String] Dog
```

The display of each context includes:

- an indicator of the active context, a preceding ==>
- the 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**.

To see source code around the selected frame, use the listw command. This requires setting the listwindow size, which defines the number of lines of source code to display.

For example:

```
topaz 1> set listwindow 3
topaz 1> listw
   name: newValue
 * ^1                          *******
```

```
          name := newValue
```

The **stack** command provides additional information about the instance and temporary variable names and values for each context.

```
topaz 1> stack
==> 1 Animal >> name:                            @1 line 1 [methId 25534209]
    receiver [25517313 sz:3 cls: 27556097 Animal] a Animal
      name                [20 sz:0 cls: 76289 UndefinedObject] nil
      favoriteFood        [20 sz:0 cls: 76289 UndefinedObject] nil
      habitat             [20 sz:0 cls: 76289 UndefinedObject] nil
    newValue [25481729 sz:3 cls: 74753 String] Dog
2 Executed Code                                @3 line 1   [methId
25504513]
    receiver [20 sz:0 cls: 76289 UndefinedObject] nil
3 GsNMethod class >> _gsReturnToC              @1 line 11 [methId 4912641]
    receiver [20 sz:0 cls: 76289 UndefinedObject] nil
```

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.

c
Same as **continue**.

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.

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. Consider the call stack we examined earlier, with **level 0** and **omit oops**:

```
topaz 1> stack
==> 1 Animal >> name:                               @1 line 1
    receiver anAnimal
    newValue Dog
2 Executed Code                                 @3 line 1
    receiver nil
3 GsNMethod class >> _gsReturnToC              @1 line 11
    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:                               @1 line 1
    receiver anAnimal
    newValue 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:                               @1 line 1
==> 2 Executed Code                                 @3 line 1
3 GsNMethod class >> _gsReturnToC              @1 line 11
```

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

Multiple Call Stacks

By default, when you continue executing code and encounter another breakpoint, the original call stack is lost.

The Topaz command **stack save** lets you retain the previous stack. This needs to be invoked for each stack you want to save.

The Topaz command **stack all** lets you display your list of saved call stacks. This display includes the top context of every call stack:

```
topaz 1> stack all
 0:  1 Animal >> habitat                      @1 line 1
 1:  1 AbstractException >> _signalWith:     @6 line 25
*2:  1 Executed Code                          @3 line 1
```

The asterisk (*) indicates the active call stack, if one exists. If there are no saved stacks, a message to that effect is displayed.

When you type the **stack change** command, Topaz sets the active call stack to the call stack indicated by the integer in the **stack all** command output, and displays the newly selected call stack:

```
topaz 1> stack change 1
Stack 1 , GsProcess 27447553
1 AbstractException >> _signalWith:        @6 line 25
```

3.3 Debugging from a different session

Normally, you will debug problems that you encounter within your running topaz session. However, you can also connect to the Gem that has the problem (the executing session) from another topaz session (the debugging session), so you can debug a problem in a session that does not have a console, such as code executing in a script.

To do this, the executing session, an RPC or linked session started from topaz or from another environment, starts a separate thread in the VM that listens for a debug connection, and provides the debug token, *random32BitDebugToken*, which is a random 32-bit integer. In a separate topaz environment, use the **debuggem** command with the PID of the executing Gem session and the debug token, to attach to the executing Gem process.

There are several ways to handle this in the executing session; `waitForDebug` encapsulates multiple functions in the executing session, but these operations can also be done separately.

Note that the `waitForDebug` handling is designed to operate where normal debugging is not possible. If you try out `waitForDebug` in interactive linked topaz, it is handled as a normal exception, which is much easier to debug, rather than enabling **debuggem** handling.

waitForDebug

The method `System >> waitForDebug` can be invoked in the Smalltalk code that will be executed; this handles the various steps required to setup for another session to attach. This can be included anywhere in your code, such within a exception handler.

For example, the executing session could run the following code:

```
[ 3 / 0]
   on:Error
   do: [:ex | System waitForDebug]
```

The `waitForDebug` method enables the remote debugging thread, and then waits for a debugging session to attach. When the error occurs, the session's PID and the *random32BitDebugToken* token are printed to the gem log (for an RPC login) or to stdout of a non-interactive linked topaz. The line will look similar to:

```
Listening for debug: DEBUGGEM 2323342 17959702887437659363
```

This DEBUGGEM expression includes all the required arguments, and is executed in the environmen tin which you will debug, such as a separate topaz environment. The debuggem command does the login in, You do not need to login explicitly.

```
topaz> DEBUGGEM 2323342 17959702887437659363
successful attach
ERROR 2706 , a Break occurred (error 2706)
topaz 2>
```

If the attach is successful, a SoftBreak or HardBreak is signaled to the executing session, which interrupts the wait loop in `System class >> waitForDebug`. This break is delivered to the debugging session, which assumes control over the executing session.

After the attach, the executing session prints a message to stdout:

```
Yielding control to Debugger Thread
```

The debugging Gem can start debugging; for example, examine the stack using **where**:

```
topaz> DEBUGGEM 2323342 17959702887437659363
successful attach
ERROR 2706 , a Break occurred (error 2706)
topaz 2> where
==> 1 Break (AbstractException) >> _defaultAction   @6 line 6
2 Break (AbstractException) >> _signal          @2 line 20
3 Break (AbstractException) >> signalToGci      @3 line 7
4 Break class (AbstractException class) >> signalToGci @3 line 7
5 System class >> waitForDebug                  @5 line 29
6 [] in Executed Code                           @8 line 3
7 ZeroDivide (AbstractException) >> _executeHandler: @7 line 11
8 ZeroDivide (AbstractException) >> _signal     @1 line 2
9 ZeroDivide (AbstractException) >> signal      @2 line 47
10 SmallInteger (Number) >> _errorDivideByZero   @6 line 7
11 SmallInteger >> /                             @6 line 7
12 [] in Executed Code                           @5 line 1
13 ExecBlock0 (ExecBlock) >> on:do:              @3 line 44
14 Executed Code                                 @2 line 2
15 GsNMethod class >> _gsReturnToC               @1 line 11
  [GsProcess 4363521]
```

Debugging

In addition to examining stack, you can:

- Use the **step** command to step through code. The waitForDebug in the executing session is cancelled.
- The **continue** command cancels the waitForDebug in the executing session, so it can continue running, without detaching the debugging session.

When you have finished debugging, use the topaz commands in the debugging session:

- the **kill** command will terminate the executing session; this also logs out the debugging session.
- the **detach** and **logout** commands detach and log out the debugging session, but leave the executing gem in waitForDebug. **debuggem** can be executed a second time to reconnect.
- the **resume** command detaches and logs out the debugging session, and cancels the waitForDebug.

Manual setup of remote debugging

There are other configuration parameters, commands and methods that allow you to setup remote debugging. The debugging thread must be enabled for the executing session, and for debugging, the executing session should be in the code you wish to debug. If you are not using `waitForDebug`, you will also need to explicitly set the process stack in the debugging session.

Enabling the debugging thread

The Gem configuration parameter GEM_LISTEN_FOR_DEBUG can be set to true for a specific Gem or Gems, either in arguments or in a configuration file. This enables the debugging thread for the Gem on login, and prints the DEBUGGEM parameters. For example, using

```
topaz -l -C 'GEM_LISTEN_FOR_DEBUG=TRUE'
```

You may also invoking `System class>>listenForDebugConnection` in the executing session, which will enable the debugging thread. This also prints the prints the DEBUGGEM parameters.

Determining the OOP of executing session's process

The debugging session must explicitly load the stack object from the executing session (the GsProcess associated with the error condition to be debugged).

Provided that **display oops** is set in topaz, the GsProcess is printed at the end of a stack displayed using (for example) **stk** or **where**. Look for a line such as:

```
[GsProcess 42086145]
```

Executing session waiting for debug

For debugging a particular issues, usually you will want the executing session to be in a known place, and wait there for the debugging session to attach.

The command **topazwaitfordebug** can be used, for example:

```
topaz> iferror topazwaitfordebug
```

Example

The following example shell script include the setup required for the executing session; debugging is enabled, and on an error the stack is printed and then the session will pause and wait for the remote debugger to attach.

```
topaz -il -C 'GEM_LISTEN_FOR_DEBUG=TRUE' << EOF
   set user DataCurator password swordfish gemstone gs64stone
   login
   display oops
   iferr 1 where
   iferr 2 topazwaitfordebug
   exec { 'foo' } at: 3 %
   logout
   exit
EOF
```

When this shell script is executed, the executing topaz linked session will encounter the index-out-of-bounds error, print the error information, and wait for the debugging session to attach.

In the executing session:

```
<startup details>
Listening for debug: DEBUGGEM 21213 10219491105650816486
successful login
topaz 1> display oops
topaz 1> iferr 1 where
topaz 1> iferr 2 topazwaitfordebug
topaz 1> exec { 'foo' } at: 3  %
ERROR 2003 , a OffsetError occurred (error 2003),
reason:objErrBadOffsetIncomplete, max:1 actual:3
topaz > exec iferr 1 : where
==> 1 OffsetError (AbstractException) >> _signalToDebugger @10
line 8
2 OffsetError (AbstractException) >> defaultAction @2 line 18
3 OffsetError (AbstractException) >> _defaultAction @4 line 4
4 OffsetError (AbstractException) >> _signal    @2 line 20
5 OffsetError (AbstractException) >> signal     @2 line 47
6 Array (Object) >> _error:args:                @15 line 11
7 Array (Object) >> _errorIndexOutOfRange:      @2 line 6
8 Array >> at:                                  @4 line 12
9 Executed Code                                 @2 line 1
10 GsNMethod class >> _gsReturnToC               @1 line 11
  [GsProcess 42086145]
topaz > exec iferr 2 : topazwaitfordebug
08/05/2023 16:45:37.423 PST
 Waiting for debugger to attach, topaz process 21213 gem
process 21213
```

Now you can start the separate debugging topaz.

In the debugging session, execute the **debuggem** command, followed by **stack set**. The information needed for these commands is in bold in the above output from the executing session.

Startup topaz and attach to the executing session:

```
topaz> debuggem 21213 10219491105650816486
successful attach
topaz 1> stack set @42086145
```

At this point you can get the stack trace and examine variables.

```
topaz 1> where
==> 1 OffsetError (AbstractException) >> _signalToDebugger @10
line 8
2 OffsetError (AbstractException) >> defaultAction @2 line 18
3 OffsetError (AbstractException) >> _defaultAction @4 line 4
4 OffsetError (AbstractException) >> _signal    @2 line 20
5 OffsetError (AbstractException) >> signal     @2 line 47
6 Array (Object) >> _error:args:                @15 line 11
7 Array (Object) >> _errorIndexOutOfRange:      @2 line 6
8 Array >> at:                                  @4 line 12
9 Executed Code                                 @2 line 1
10 GsNMethod class >> _gsReturnToC               @1 line 11
```

If the connection is successful, the debugging session will have debugging control over the executing session, and debugging can be done as described on page 63.

Chapter
4
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**, and so on. Commands that cannot be abbreviated are described in the individual command documentation.

If a command abbreviation is ambiguous, it is not executed. Note however that if a command's first letters are abbreviated and this matches another command, the other command is executed; for example, the **l** form of **listw**, and the **c** form of **continue**.

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 38.

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 snapshot view of the repository data.

This command cannot be abbreviated.

ALLSTACKS

Print the stacks of all instances of GsProcess that are known to the ProcessorScheduler instance in the VM and stacks associated with previous topaz **stack save** commands.

See also **threads** on page 192.

BEGIN

Begins a GemStone transaction. If the session is already in a transaction, this has the effect of an abort. The **begin** command is only useful if your session is not in automatic transaction mode, i.e., in manual or transactionless transaction mode.

You can change transaction mode using the method System class >> transactionMode: with an argument of #autoBegin, #manualBegin, or #transactionless, or by using the topaz **set transactionmode** command (page 175).

This command cannot be abbreviated.

Example

```
topaz 1> set transactionmode manualbegin
transaction aborted and mode changed to manualBegin
topaz 1> begin
<perform database operations>
topaz 1> commit
Successful commit
```

BREAK

break *aSubCommand*

Establishes or displays a method breakpoint within GemStone Smalltalk code. Subcommands are **method**, **classmethod**, **list**, **enable**, **disable**, and **delete**. For more information about using breakpoints, see Chapter 3, "Debugging Your GemStone Smalltalk Code".

Setting Breakpoints

You can set breakpoints on a method, or at any step point within a method. Step points includes assignments, message sends, and method returns. To display the step points for a method, use the **list steps** command.

In the following commands, the argument @ *stepPoint* specifies the step point within that method where the break is to occur. If you omit the step point, 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 can specify the method using method/classmethod, or using the >> syntax.

break method *classSpecification methodName* [@ *stepPoint*]

break classmethod *classSpecification methodName* [@ *stepPoint*]

break methodd *methodSpecification* [@ *stepPoint*]

break classmethod *methodSpecification* [@ *stepPoint*]

break @ *stepPoint*

classSpecification can be defined in these ways:

className	The name of the class that implements *methodName*.
`@`*integer*	An unsigned 64-bit decimal OOP value that denotes the class.
`**`	The class that was the result of the last execution.
`^`	The current class, as defined by the most recent `set class` (or other command that sets the current class)

methodSpecification is an unsigned 64-bit decimal OOP value of a GsNMethod instance, which defines both the class and method.

The @ *stepPoint* format applies after certain other commands that set a specific selected method, such as **lookup** and **list**.

The **break** command accepts the >> syntax to specify a method, using the following forms:

break *className* **>>** *methodName* [@ *stepPoint*]

break *className* **class >>** *methodName* [@ *stepPoint*]

break *className* **(***implementationClassName***) >>** *methodName* [@ *stepPoint*]

break *className* **class** (*implementationClassName*) **>>** *methodName* [@ *stepPoint*]

The *implementationClassName* specifies that the method lookup begins with *implementationClassName*, normally a superclass of *className*. This style is used in topaz stack summaries, and allows paste in of lines from the stack.

Examples

The following expressions set breaks in the same method, the KeyValueDictionary instance method `at:put:`. The same patterns apply for class methods.

```
break KeyValueDictionary >> at:put: @ 1

break @79361 >> at:put: @ 2

break IdentityDictionary (KeyValueDictionary) >> at:put: @ 3

break method KeyValueDictionary at:put: @ 4

break method @79361 at:put: @ 5

set class KeyValueDictionary
break method ^ at:put: @ 7

exec KeyValueDictionary %
break method ** at:put: @ 6

break method @5094401 @ 8

lookup KeyValueDictionary >> at:put:
break @9
```

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

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

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 as the first character in the line. 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.

This command cannot be abbreviated.

CONTINUE/C

continue [*anObjectSpec*]

c [*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.

`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 38.

For more information about breakpoints, see the discussion of the **break** command on page 71, or see Chapter 3, "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 *Programming Guide for GemStone/S 64 Bit*. User manuals for the GemStone interfaces, such as *GemBuilder for Smalltalk*, also contain discussions of error-handling.

DEBUGGEM

debuggem *processId token*

Takes two integer arguments, a processId and a token.

The processId must be the PID of a gem or topaz -l, running on localhost, which was either configured with GEM_LISTEN_FOR_DEBUG=true, or which has previously executed `System class >> listenForDebugConnection`.

The token is a random integer printed by that target gem process to its log file (or stdout and current .out file of the topaz -l), or returned by `System class >> listenForDebugConnection` or `System class >> waitForDebug`.

debuggem attempts to attach to the target gem and create a topaz session in this topaz process which has debugging control over the target gem.

If Smalltalk execution is in progress in the target gem, it is interrupted with equivalent of GciSoftBreak() and the resulting stack can be examined by the topaz session created by **debuggem**.

It is recommended that an error handler invoke `System class >> waitForDebug` to wait for topaz to attach with **debuggem**.

In a topaz session created with **debuggem**,

- **kill** will execute kill -TERM against the topaz -l or gem being debugged.
- **continue**executes System cancelWaitForDebug and stays attached, where it can handle other errors or breakpoints.
- **detach** and **logout** both execute GciLogout and leave the gem in waitForDebug; you can execute **debuggem** again to reconnect.
- **resume** executes System cancelWaitForDebug and GciLogout.
- **step** executes System cancelWaitForDebug before performing the single step.

If execution is not in progress, then the **debuggem** topaz gets control of the target gem, blocking further GCI calls from the target gem's client or blocking further topaz commands from the target's topaz -l.

If a fetch or store traversal is in progress, the **debuggem** topaz will get control of the target gem when the traversal completes.

debuggem cannot be abbreviated, it must be typed in full.

For more information, see "Debugging from a different session" on page 62.

The following commands are disallowed when attached using **debuggem**:

- **debuggem**
- **define**
- **exitifnoerror**
- **expectvalue**
- **expecterror**
- **expectbug**
- **fileout**
- **fileformat**
- **gcitrace**
- **iferr**
- **iferr_clear**
- **iferr_list**
- **login**
- **loadua**
- **logoutifloggedin**
- **nbrun**
- **nbstep**
- **nbresult**
- **pausefordebug**
- **pollforsignal**
- **protectmethods**
- **unprotectmethods**
- **removeallclassmethods**
- **removeallmethods**
- **shell**
- **stackwaitfordebug**
- **topazwaitfordebug**
- **topazpausefordebug**

DEBUGRUN

Like **run**, but sets flags so that execution will stop at the first step point within the source text. The text following the **debugrun**, and up to the first line that contains a % as the first character in the line, is sent to GemStone for execution as GemStone Smalltalk code.

Execution stops at the first step point within this code, and you may use the **step** command (page 182) to step through execution.

For example:

```
topaz 1> debugrun
Time now
%
a Breakpoint occurred (error 6002), Single-step breakpoint encountered.
1 1 Executed Code                                 @1 line 1
topaz 1> step
a Breakpoint occurred (error 6002), Single-step breakpoint encountered.
1 1 Executed Code                                 @2 line 1
topaz 1> step into
a Breakpoint occurred (error 6002), Single-step breakpoint encountered.
1 1 Time class >> now                             @2 line 1
topaz 1> step into
a Breakpoint occurred (error 6002), Single-step breakpoint encountered.
1 1 Time class >> now                             @3 line 9
topaz 1> step into
a Breakpoint occurred (error 6002), Single-step breakpoint encountered.
1 1 DateAndTime class (DateAndTimeANSI class) >> now @2 line 1
...
```

See Chapter 3, "Debugging Your GemStone Smalltalk Code", for more information on breakpoints and debugging code.

debugrun cannot be abbreviated, it must be typed in full

DEFINE

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 38) are legal in **define** commands. The variable name *aVarName* must begin with a letter (a..z or A..Z) or an underscore, can be up to 255 characters in length, and cannot contain white space.

You may not redefine the topaz predefined variables CurrentMethod, ErrorCount, CurrentCategory, CurrentClass, LastResult, LastText, and myUserProfile.

`define`
Lists all current local variable definitions.

`define` *aVarName*
Deletes the definition of the variable *aVarName*.

`define` *aVarName anObjectSpec*
Define a local variable whose value is result of *anObjectSpec*.

`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*.

For example:

```
topaz 1> define CurrentSessions System currentSessionNames
topaz 1> define UserId myUserProfile userId
topaz 1> define
Current definitions are:
  UserId             = 2673409
  CurrentSessions    = 33659905
------------------------
  CurrentMethod      = nil
  ErrorCount         = 2
  CurrentCategory    = nil
  CurrentClass       = nil
  LastResult         = nil
  LastText           = nil
  myUserProfile      = 1458177
```

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

DETACH

For use when debugging a second process from another topaz session, using the **debuggem** command (page 78).

In the debugging topaz session (the session created by **debuggem**), the **detach** command disconnects from the executing gem or linked process (the session being debugged), executes the equivalent of

```
   System cancelWaitForDebug
```

and logs out. The GsProcess being debugged will be continued after the logout, and `System class >> waitForDebug` will return to the caller.

DISASSEM

disassem [*aClassParameter*] *aParamValue*

The **disassem** command allows you to disassemble the specified GsNMethod, displaying the bytecode details. This command is intended for use in linked (**topaz -L** or **topaz -l**). With an RPC session, output goes to 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.

```
topaz 1> set class System
topaz 1> disassem classmethod: gemEnvironmentVariable:
System class >> gemEnvironmentVariable:  method:10029057
, literals at IP 88
Step/Source IP    Opcode         Description  (IP in bytes from
&obj.hdr)
 ---/------ ---   -------------- -----------
  1/     1    56    CHKInt
              58    PUSH_LITERAL_oldGen_u1 u1=1e flag=0 zlitIdx:8
literal:Oop 135169 aClass:  GsFile
              64    PUSH_STK_s1 s1=16 bytes(2 words)
              68    PUSH_SPECIAL_OOP_s1 s1=12 val=Oop 12 Class:
Boolean
  2/   536    72    SEND_u1_u32 sel=Symbol(oop 5768961)
#'_expandEnvVariable:isClient:' env:0, 2 args, sendCache.u2=24
  3/   527    80    RETURN_TOS_u1 u1=24 bytes(3 words)
```

DISPLAY

display *aDisplayFeature*

The **display** and **omit** commands control the display of Gemstone Smalltalk objects and other features related to output.

The **display** command turns on these display attributes, and the **omit** command turns them off.;

alloops

`display alloops`

If **display alloops** is set, enables the display of OOPs of classes along with class names in object display, and displaying the oops of primitive specials. 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.

Default: **omit alloops**

bytes

`display bytes`

When displaying byte-format objects such as Strings and ByteArrays, include the hexadecimal value of each byte.

Default: **omit bytes**

for example,

```
topaz 1> exec #[ 97 98 99 ] %
  1 'abc'                      61 62 63
```

classoops

`display classoops`

Legacy; equivalent to **display alloops**.

decimalbytes

`display decimalbytes`

When displaying byte-format objects such as Strings and ByteArrays, include the decimal value of each byte.

Default: **omit decimalbytes**

for example,

```
topaz 1> exec #[ 97 98 99 ] %
 1 'abc'                       97  98  99
```

deprecated

`display deprecated`

For topaz commands such as **strings** and **senders**, which return lists of methods, display deprecated causes those results to include methods which contain sends of `deprecated`, `deprecated:`, or `deprecatedNotification:`.

Default: **display deprecated**

Deprecated methods are those that send one of the following messages:

```
deprecated
deprecated:
deprecatedNotification:
```

For example, the method `concurrencyMode` is deprecated:

```
topaz 1> implementors concurrencyMode
System class >> concurrencyMode
topaz 1> omit deprecated
topaz 1> implementors concurrencyMode
(Omitted 1 deprecated methods)
```

flushoutput

`display flushoutput`

enables immediate flushing of topaz output files other than stdout. **display flushoutput** causes an fflush() call to be made to each active topaz log file (files specified by **output push**) other than stdout, after each line of topaz output is written. This allows a tail -f to have a more up-to-date view of a topaz output file.

Default: **omit flushoutput**

errorcheck

`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.

Default: **omit errorcheck**

lineeditor

`display lineeditor`

Enables the use of the Topaz line editor, using the open source linenoise library. This is not available on Windows.

The number of lines of history kept for the lineeditor is controlled using the **set history** command (page 171).

The **status** command includes one of the followings lines to indicate if the lineeditor is in use:

```
using line editor
not using line editor
```

Default: **display lineeditor**

oops

`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.

The oops of certain primitive special objects, such as Characters, SmallIntegers, Booleans, and nil, do not display their oops. To enable display of their oops, use **display alloops**.

Default: **display oops**

names

`display names`

For each of an object's named instance variables, displays the instance variable name along with its value.

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

Default: **display names**

pauseonerror

`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.

The **status** command includes one of the followings lines to indicate status of this attribute:

```
display interactive pause on errors
omit interactive pause on errors
```

Default: **omit pauseonerror**

pauseonwarning

`display pauseonwarning`

When a compiler warning occurs, and if the topaz stdin is from a terminal, then write a message "Pausing after warning ..." to stdout, and wait for an end of line on stdin before continuing topaz execution. A ctl-C on stdin will terminate the pause and terminate further processing of input files.

The **status** command includes one of the followings lines to indicate status:

```
display interactive pause on warnings
omit interactive pause on warnings
```

Default: **omit pauseonwarning**

pushonly

`display pushonly`

Enables the effect of the **only** keyword in an **output push** command. Does not effect output to stdout within a topaz -S script. To disable this effect, use the **omit pushonly** command.

Default: **display pushonly**

resultcheck

`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.

Default: **omit resultcheck**

singlecolumn

`display singlecolumn`

rather than displaying large numbers (more than eight) of method and temporary variable definitions with a debugger stack frame in columns, display in a single column.

Default: **omit singlecolumn**

stacktemps

`display stacktemps`

enables the display of stack frames to include un-named evaluation temps which have been allocated by bytecodes within the method.

Default: **omit stacktemps**

versionedclassnames

When **display versionedclassnames** is set, which is the default, topaz includes [*verionNumber*] following the class name, for instances of classes that are not the last one in the classHistory of that class. This is limited to when **display oops** is also set.

zerobased

`display zerobased`

Shows offsets of instance variables as zero-based when displaying objects. With **display zerobased**, offsets are zero-based, with **omit zerobased**, offsets are one-based.

Default: **omit zerobased** (offsets are one-based, as in Smalltalk) .

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 `%` as the first character in the line. 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 *Programming Guide for GemStone/S 64 Bit*.

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*.

Note that **doit** always displays results at level 0, regardless of the current display level setting (page 123). The **doit** command does not alter the current level setting.

DOWN

down [*anInteger*]

Moves the active frame down within the current stack, and displays the frame selected as a result. The optional argument *anInteger* specifies how many frames to move down. If no argument is supplied, the scope will go down one frame. See also **stack down** on page 179.

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

```
topaz 1> where
1 ZeroDivide (AbstractException) >> _signalWith: @6 line 25
2 ZeroDivide (AbstractException) >> signal      @2 line 47
3 SmallInteger (Number) >> _errorDivideByZero   @6 line 7
==> 4 SmallInteger >> /                            @6 line 7
5 [] in  Executed Code                          @2 line 1
6 Array (Collection) >> do:                     @5 line 10
7 Executed Code                                 @2 line 1
8 GsNMethod class >> _gsReturnToC               @1 line 11

topaz 1> down
3 SmallInteger (Number) >> _errorDivideByZero   @6 line 7
    receiver 1

topaz 1> where
1 ZeroDivide (AbstractException) >> _signalWith: @6 line 25
2 ZeroDivide (AbstractException) >> signal      @2 line 47
==> 3 SmallInteger (Number) >> _errorDivideByZero   @6 line 7
4 SmallInteger >> /                             @6 line 7
5 [] in  Executed Code                          @2 line 1
6 Array (Collection) >> do:                     @5 line 10
7 Executed Code                                 @2 line 1
8 GsNMethod class >> _gsReturnToC               @1 line 11
```

DUMPOBJ

dumpobj *anObjectSpecification*

This command does a low-level dump of an object, the node of a large object, or the node of an IdentityBag. For a byte format object, it displays the integer values of the bytes. This command cannot be abbreviated

For example

```
topaz 1> dumpobj Globals
aSymbolDictionary
  #1 657
  #2 112
  #3 20260
  #4 1013
  #5 nil
  #6 anIdentityCollisionBucket
  #7 ReadWriteStreamPortable
  #8 aSymbolAssociation
  #9 nil
  #10 anIdentityCollisionBucket
  #11 PositionableStreamPortable
  #12 aSymbolAssociation
  #13 nil
  #14 anIdentityCollisionBucket
  #15 ReadStreamLegacy
  #16 aSymbolAssociation
   ...
```

EDIT

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` *className* **`>>`** *selectorSpec*
`edit` *className* **`class >>`** *selectorSpec*

Allows you to edit the source code of an existing instance or class method. For example:

```
topaz 1> edit Animal >> habitat
topaz 1> edit Animal class >> name:species:habitat:
```

`edit method:` *selectorSpec*

Edit the source code of an existing instance method. Before you can use this subcommand, 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
```

`edit classmethod:` *selectorSpec*
Edit the source code of an existing class method. Before you can use this subcommand, 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.

`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`."

ENV

env [*anInteger*]

Sets the compilation environmentId used for method compilations, lookups of senders and implementors, and **run**, **printit**, etc. *anInteger* must be between 0 and 255 and is 0 by default. Values other than 0 are invoked in additional execution environments and are used in specialized applications.

With no arguments, prints the current compilation environmentId.

The compilation environment may also be set using the commend **set compile_env:**.

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.

```
topaz> errorcount
0
```

It is equivalent to

```
topaz 1> object ErrorCount
```

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

This command cannot be abbreviated.

EXEC

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

The **exec** command, unlike related commands such as **run** and **printit**, accept text on the same line as the **exec** command itself, up to a % character on that line. If there is no % on the exec command line, subsequent lines are included as part of the Smalltalk code to be executed, up to a % character appearing as the first character in a line.

For example:

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

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

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

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*.

Like **run**, **exec** uses the current display level setting (page 123).

EXIT

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.

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. Only 8 bits of the integer are returned. To get valid return values, the argument should always resolve to a value in the range 0..255.To use an object specification, including errorcount, you must be logged in when the **exit** command is executed.

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).

For example:

```
topaz 1> exec ( 42 - 3) %
topaz 1> exit **
Logging out session 1.
--- 08/11/2023 14:11:53.619 PDT Logging out
unix> echo $?
39
```

exit is ignored when reading a file using **input**, when stdin is a tty. If topaz stdin is redirected to a file and the file does not end with a **quit** or **exit**, topaz considers EOF on stdin to be an error, and will result in a non-zero topaz exit status.

This command cannot be abbreviated.

To exit from topaz with an exit status directly from Smalltalk code, you can use the ExitClientError class. For example,

```
topaz 1> run
ExitClientError
      signal: 'Disallowed Operation'
      status: 34
%
ERROR 3004 , a ExitClientError occurred (error 3004), ,
   Disallowed Operation (ExitClientError)
Logging out session 1.
--- 08/11/2023 15:21:17.137 PDT Logging out
```

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** (described on page 96). Otherwise, this command has no effect.

This command cannot be abbreviated.

EXPECTBUG

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 (described on page 102).

errorCategory and *errNumCls*
are specified as in the **expecterror** command (described on page 99).

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
```

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

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**.You must also use it before executing **continue** after a breakpoint.

anErrorCategory must be a Topaz object specification that evaluates to the object identifier of an error category; normally, GemStoneError.

anErrorNumCls must be a Topaz object specification that evaluates either to a SmallInteger legacy error number, or to the object identifier of a subclass of Abstract Exception.

All Topaz object specification formats (as described in "Specifying Objects" on page 38) are legal in **expecterror** commands.

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.

```
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
```

If execution returns unanticipated results, Topaz prints a message (in this example, "ERROR: WRONG CLASS of Exception"), 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, attempt to divide 1 by zero
(ZeroDivide)
ERROR: WRONG CLASS of Exception, does not match expected class
topaz > exec iferr 1 : where
==> 1 ZeroDivide (AbstractException) >> _signalWith: @5 line 25
2 ZeroDivide (AbstractException) >> signal      @2 line 47
3 SmallInteger (Number) >> _errorDivideByZero   @6 line 7
4 SmallInteger >> /                             @6 line 7
5 Executed Code                                 @2 line 1
6 GsNMethod class >> _gsReturnToC               @1 line 11
topaz 1> errorcount
1
```

Further arguments to EXPECTERROR

In addition to the error category and class, you may optionally specify additional arguments. The **expecterror** argument values are tested against the argument values returned with the error. If one or more of the argument values do not match, errorcount is incremented. .

In additional to standard object specification formats, you may use additional formats 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 specified as a String or Symbol (enclosed in single quotes and/or with a leading #), then it will be regarded as matching if the result if the two are equal (=).

Otherwise, Topaz regards it as matching the result if the two are identical (==).

You may omit arguments, which will not count as an error. If you specify more **expecterror** arguments than the actual error returns, then errorcount will be incremented. To match any error argument, use /Object.

For example:

```
topaz 1> errorcount
2
topaz 1> expecterror GemStoneError OffsetError %Array 3 6
topaz 1> run
(Array new: 3) at: 6
%
ERROR 2003 , a OffsetError occurred (error 2003),
reason:objErrBadOffsetIncomplete, max:3 actual:6 (OffsetError)
topaz 1> errorcount
2
```

EXPECTVALUE

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**. You must also use it before executing **continue** after a breakpoint.

All Topaz object specification formats (as described in "Specifying Objects" on page 38") 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 *anObjectSpec* is specified as a String or Symbol (enclosed in single quotes and/or with a leading #), then it will be regarded as matching if the result if the two are equal (=).

Otherwise, 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
```

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> errorcount
0
topaz 1> iferror stack
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
```

FILEFORMAT

fileformat 8bit | utf8

This command controls the interpretation of Character data for input and fileout, to allow strings containing Characters with codepoints over 255 to be input and output.

This is meaningful if you are using text that contains any Characters with values over 127. Characters 127 and below are 7-bit, and the code points are the same as the UTF-8 encoded values, and so are not affected by this setting.

Characters in the range of 128-255 can be read and written with their 8-bit codepoints, or read and written encoded as UTF-8; these produce different results. So if such text is written as UTF8, it must be read in with a fileformat of UTF8 in order to get correct results, and similarly both written and read as 8-bit in order to recreate the same text.

To avoid misinterpretation of fileouts, the **fileout** command writes a fileformat command at the start of the fileout. A **fileformat** command within a file only has effect within that file and any nested files.

Note that this setting does not apply to files produced by the **output** command. **output push**, etc. write files in UTF-8 format, regardless of the setting for **fileformat**.

The following options are supported:

utf8

```
fileformat utf8
```

Sets the fileformat to UTF-8. Code that is filed out using **fileout** is encoded in UTF-8, and files read using **input** are interpreted as being UTF-8 and are decoded accordingly.

8bit

```
fileformat 8bit
```

Sets the fileformat to 8-bit, for compatibility with older releases. Code that is filed out using **fileout** are written as bytes, not encoded as UTF-8. Fileout of code containing Characters with codePoints over 255 will error.

The default at topaz startup is utf8. If you are filing in code from an older release, verify that the header includes the fileformat 8bit, or explictly set this in your topaz session.

Input from stdin that is a tty is always interpreted as UTF-8; changing the **fileformat** of a tty stdin to **8bit** is not allowed.

FILEOUT

fileout [*command*] *clsOrMethod* [tofile: *filename* [format: *fileformat*]]

Writes out class and method information in a format that can be fed back into Topaz with the **input** command. Subcommands are used to specify whether to file out the entire class, or specific method or methods. If none of the defined subcommands follow the **fileout**, then the next word is assumed to be a selector for an instance method on the current class.

By default, the fileout command outputs the fileout text to stdout. To direct this to a file, follow the specification of what to fileout with 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.

When using the **tofile:** keyword, you may also optionally specify the format: keyword. This must be either 8bit or UTF8, and specifies whether the file is written out in bytes, or encoded in UTF-8. This overrides the current topaz setting for **fileformat**.

All fileout output generated from the **fileout** command include commands setting the **fileformat** and **set sourcestringclass**, based on the current settings or the **format:** command.

`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.

FR_1

fr_1 [*anInteger*]

Similar to the **frame** command (described on page 108), but for large numbers (more than eight) of method and temporary variables, these are displayed one per line rather than in four columns.

This command cannot be abbreviated.

FR_CLS

fr_cls [*anInteger*]

Similar to the **frame** command (described on page 108), but also displays OOPs of classes along with class names in the specified stack frames.

This command cannot be abbreviated.

FRAME

frame [*anInteger*]

Moves the active frame to the frame specified by *anInteger*, within the current stack, and displays the frame selected as a result. The display includes parameters and temporaries.

If no argument is supplied, displays the current frame.

See also **stack scope** on page 178, the **up** command on page 197 and the **down** command on page 89.

For example:

```
topaz 1> printit
{ 1 . 2} do: [:x | x / 0 ]
%
ERROR 2026 , a ZeroDivide occurred (error 2026),
reason:numErrIntDivisionByZero, An attempt was made to divide 1 by
zero. (ZeroDivide)

topaz 1> where
==> 1 ZeroDivide (AbstractException) >> _signalWith: @6 line 25
2 ZeroDivide (AbstractException) >> signal      @2 line 47
3 SmallInteger (Number) >> _errorDivideByZero   @6 line 7
4 SmallInteger >> /                             @6 line 7
5 [] in  Executed Code                          @2 line 1
6 Array (Collection) >> do:                     @5 line 10
7 Executed Code                                 @2 line 1
8 GsNMethod class >> _gsReturnToC              @1 line 11

topaz 1> frame 4
4 SmallInteger >> /                             @6 line 7
    receiver 1
    aNumber 0

topaz 1> where
1 ZeroDivide (AbstractException) >> _signalWith: @5 line 25
2 ZeroDivide (AbstractException) >> signal      @2 line 47
3 SmallInteger (Number) >> _errorDivideByZero   @6 line 7
==> 4 SmallInteger >> /                             @6 line 7
5 [] in  Executed Code                          @2 line 1
6 Array (Collection) >> do:                     @5 line 10
7 Executed Code                                 @2 line 1
8 GsNMethod class >> _gsReturnToC              @1 line 11

topaz 1> frame 7
7 Executed Code                                 @2 line 1
    receiver nil
```

GCITRACE

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.

This command cannot be abbreviated.

HELP

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

hierarchy [*aClassName*] [*environmentId*]

Prints the class hierarchy up to Object for the specified class, in the specified environmentId, or the current environmentId if no environmentId is specified.

If you do not specify a class, Topaz prints the hierarchy for the current class.

Example

```
topaz 1> hierarchy SmallFraction
Object
  Magnitude
    Number
      AbstractFraction
        SmallFraction
```

See Also

subhierarchy (page 187)

HISTORY

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 168) establishes the maximum number of command lines to retain in the Topaz line editor history.

Example

```
topaz 1> history
0 login
1 run
2 2 / 0
3 %
4 stk
5 display oops
6 stk
7 frame 3
8 history
```

IFERR

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 an integer 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. This allows you to execute a number of handling commands. For example:

```
topaz 1> iferr 1 where
topaz 1> iferr 2 allstacks
topaz 1> iferr 3 topazwaitfordebug
```

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 examples demonstrate how to use **expecterror** to test the kind of error that is returned from Smalltalk code execution, so you can conditionally avoid triggering the **iferr** command.

The following reports an error, but does not perform the **iferr** operations:

```
topaz 1> iferr 1 stk
topaz 1> expecterror GemStoneError MessageNotUnderstood
topaz 1> exec 2 foo %
ERROR 2010 , a MessageNotUnderstood occurred (error 2010), a
SmallInteger does not understand  #'foo'
topaz 1>
```

But in the following, since divide-by-zero is an unexpected error, does print the stack:

```
topaz 1> iferr 1 stk
topaz 1> expecterror GemStoneError MessageNotUnderstood
topaz 1> exec 2 / 0 %
ERROR 2026 , a ZeroDivide occurred (error 2026),
reason:numErrIntDivisionByZero, attempt to divide 2 by zero
ERROR: WRONG CLASS of Exception, does not match expected class
topaz > exec iferr 1 : stk
==> 1 ZeroDivide (AbstractException) >> _signalWith: @5 line 25
2 ZeroDivide (AbstractException) >> signal      @2 line 47
3 SmallInteger (Number) >> _errorDivideByZero   @6 line 7
4 SmallInteger >> /                             @6 line 7
5 Executed Code                                 @2 line 1
6 GsNMethod class >> _gsReturnToC               @1 line 11
```

This command cannot be abbreviated.

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 113.

This command cannot be abbreviated.

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 113.

This command cannot be abbreviated.

IFERROR

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 113.

This command cannot be abbreviated.

IMPLEMENTORS

implementors *selectorSpec*

Displays a list of all classes that implement the given *selectorSpec* (either a String or a Symbol). For example:

```
topaz 1> implementors asByteArray
Collection >> asByteArray
MultiByteString >> asByteArray
String >> asByteArray
```

This command is equivalent to the following:

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

This command may use significant temporary object memory. Depending on your repository, 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*.

INPUT

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.

When input commands are echoed, as they are by default, the topaz prompt indicates that the given command is located in a particular level of nested file by including a + (plus sign) for each level of file input nesting.

For example, assuming *aFile* contains a series of commands which include **import** *nestedFile*:

```
topaz 1> input aFile
topaz 1 +> <commands in aFile>
topaz 1 +> input nestedFile
topaz 1 ++> <commands in nestedFile>
topaz 1 +> <further commands in aFile>
```

`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

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.

This command is identical to the **run** command (page 163), except that the interp command does not use native code, the Smalltalk code execution is interpreted.

GemStone Smalltalk text is terminated by the first line that contains a `%` as the first character in the line. 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 cannot be abbreviated.

KILL

For use when debugging a second process from another topaz session, using **debuggem** command (page 78).

In a topaz session that was created by **debuggem** , the **kill** command executes `kill -TERM` against the gem or linked process being debugged.

LEVEL

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 10000.

Note the following:

- The **run** command (page 163) displays results using the current display level, as set by the **level** command.
- The **doit** command (page 88) always displays results at level 0, regardless of the current display level.
- The **printit** command (page 155) always displays results at level 1, regardless of the current display level.

LIMIT

limit [bytes | oops | lev1bytes | lev2oops] *anInteger*

Tells Topaz how much of any individual object to display in GemStone Smalltalk results. The display can be limited by OOPs, to control the number of objects displayed (for example, the number of elements in a collection). It can also be limited by bytes, to control the number of bytes of byte objects, such as Strings, that are displayed.

For example, limit bytes 100 would tell Topaz to only display 100 bytes of any String (or other byte object).

A limit of 0 tells Topaz to not limit the size of the output. This is the default.

If the amount that would be displayed is limited by limit bytes setting, the display indicates missing text using ...(*NN* more bytes). If the number of objects is limited by a limit oops setting, then it prints ... *NN* more instVars.

bytes

`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`.

oops

`limit oops` *anInteger*

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

lev1bytes

`limit lev1bytes` *anInteger*

When the topaz **level** is set to 1 or greater, this limit controls how many bytes to display of instVar values and frame temporaries. Default is 100. If **lev1bytes** is set to zero, then the value of "limit bytes" is used for instVar values and frame temporaries.

lev2oops

`limit lev2oops` *anInteger*

When the topaz **level** is set to 2or greater, this limit controls how many oops to display of instVar values.

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 **lookup**, **up**, **down**, or **frame** command.

Browsing Dictionaries and Classes

dictionaries

`list dictionaries`

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

classesIn:

`list classesIn:` *aDictionary*

Lists the classes in *aDictionary*. For example,

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

lists all of the classes in your UserGlobals dictionary.

classes

`list classes` [*subStringPattern*]

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

To limit the displayed classes, you can include an argument *subStringPattern*. Only results that match, case insensitive, some portion of the *subStringPattern* are displayed. For example,

```
topaz 1> list classes cert
GsX509Certificate
GsX509CertificateChain
```

categoriesIn:

`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.

icategories

`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.

ccategories:

`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.

selectors

`list selectors` *[token]*

Lists selectors of all instance methods in the current class, for which the argument *token* is a case-insensitive substring of the selector. If *token* is omited, list all instance method selectors in the current class. May be abbreviated as `list sel`.

For example,

```
topaz 1> set class Array
topaz 1> list sel at:p
    at:put:
    _at:put:
    _basicAt:put:
```

cselectors

`list cselectors` *[token]*

Lists selectors of all class methods in the current class, for which the argument *token* is a case-insensitive substring of the selector. . If *token* is omited, list all class method selectors in the current class. May be abbreviated as `list csel`.

primitives

`list primitives` *[token]*

Lists selectors of all class methods in the current class that invoke primitives. If *token* is specified, only return the selectors that contain a case-insenstive substring matching the given token string. May be abbreviated as `list prim`.

cprimitives

`list cprimitives` *[token]*

Lists selectors of all instance methods in the current class that invoke primitives. If *token* is specified, only return the selectors that contain a case-insenstive substring matching the given token string. May be abbreviated as `list cprim`.

Listing Methods

`list` with no arguments lists the source code of the active method context. See Chapter 3, "Debugging Your GemStone Smalltalk Code", starting on page 55.

`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.

method:

`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`.

classmethod:

`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:` or `list cmethod`.

imethod:

`list imethod:` is the equivalent to **`list method:`**.

linenumbers

`list linenumbers`
This is an additional option to one of the above commands, preceeding the method specification. When this is used, the line number is included in the listing. For example:

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

Listing Step Points

step

`list step`
Lists the source code of the current frame, and display only the step point corresponding to the step point of the current frame.

steps

`list steps`
Lists the source code of the current frame, 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 3, "Debugging Your GemStone Smalltalk Code", starting on page 55.

`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.

stepips

`list stepips`

list stepips, **list stepips method:**, and **list stepips classmethod:** list source code and display the IPs (instructions pointers) of the step points. These commands are intended for low-level debugging and not normally useful for customer development debugging.

Listing Breakpoints

In addition to **list breaks**, you can use the **break list** command to list all currently set breakpoints. For more information about using breakpoints, see Chapter 3, "Debugging Your GemStone Smalltalk Code", starting on page 55.

breaks

`list breaks`

Lists the source code of the current frame, 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 _stringCharSize bitAnd: 16r7) ~~ 0 ifTrue:[
     ^ (DoubleByteString withAll: self) < aCharCollection .
   ].
   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 / L

listw

l

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.

For example:

```
topaz 1> stk
==> 1 ZeroDivide (AbstractException) >> _signalWith: @5 line 25
2 ZeroDivide (AbstractException) >> signal      @2 line 47
3 SmallInteger (Number) >> _errorDivideByZero   @6 line 7
4 SmallInteger >> /                             @6 line 7
5 [] in  Executed Code                          @2 line 1
6 Array (Collection) >> do:                     @5 line 10
7 Executed Code                                 @2 line 1
8 GsNMethod class >> _gsReturnToC              @1 line 11

topaz 1> frame 4
4 SmallInteger >> /                             @6 line 7
    receiver 1
    aNumber 0

topaz 1> listw
   / aNumber

   "Returns the result of dividing the receiver by aNumber."

   <primitive: 10>
   (aNumber _isInteger) ifTrue:[
     (aNumber == 0) ifTrue: [^ self _errorDivideByZero].
 *                                ^6
*******
     ^ Fraction numerator: self denominator: aNumber
   ].
   ^ super / aNumber
```

The **listw** command cannot be abbreviated, other than by **l**.

LITERALS

literals *anObject*

Reports all methods in which *anObject* is contained as a literal reference. *anObject* is typically a String, Symbol, or Number.

literals is equivalent to:

```
topaz 1> exec ClassOrganizer new literalsReport: anObject %
```

for example,

```
topaz 1> literals TimeZone
TimeZone >> =
TimeZone class >> default:
TimeZone class >> fromLinux
```

LOADUA

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** command)

`set gemstone:`
name of the Stone (repository monitor) process, if necessary including node and protocol information in the form of a network resource string (NRS). See the *System Administration Guide* for more on NRS syntax and usage.

`set username:`
your GemStone user ID.

`set password:`
your GemStone password.

`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.

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 explicitly set, 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, you must set **gemnetid**, and 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 (obviously). 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, read the section "Logging In to GemStone" on page 14. The **set** command is described on page 168.

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.

This command cannot be abbreviated.

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 the **logout** command (page 134), 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.

This command cannot be abbreviated.

LOOKUP

lookup (meth | method | cmeth | cmethod) *selector [envId]*

lookup *className* [class] [(*implementingclass*)] >> *selector [envId]*

The **lookup** command to search upwards through the hierarchy of superclasses to locate the implementation of a given method selector. If no *envId* is specified, the current compilation environment **env** is used.

There are two ways to specify the class, selector, and if it is a class or instance method:

- using the **set** command to specify the class, and either **method** or **classmethod** to indicate a class or instance method;
- using *className* `[class] >>` to identify this information

All arguments to **lookup** are case sensitive.

The *selector* argument is string or symbol with the full method selector. The following are all acceptable *selector* specifications:

```
lookup hash
lookup 'hash'
lookup #hash
lookup #'hash'
```

Or you may use syntax such as:

```
lookup SmallInteger >> +
lookup String class >> new
```

The **lookup** command also accepts the text generated in stack frame, so you can copy and paste from a stack frame to lookup a method.

```
lookup String (SequenceableCollection) >> at:ifAbsent:
```

Using this syntax, *implementingClass* is used as the starting point for lookup, and the *class* argument is ignored.

Related commands include **senders** and **implementors**.

Finding and Listing Methods

lookup classmethod *selector*
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 *selector*
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
%
```

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

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

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

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

Pasting from stack frames

When you are stepping through code or examining the call stack for an error, topaz displays stack frames containing the individual message sends. You can cut and paste the printed methods into the lookup command, to lookup the source code that was executed.

For example:

```
topaz 1> run
1 / 0
%
ERROR 2026 , a ZeroDivide occurred (error 2026),
reason:numErrIntDivisionByZero, attempt to divide 1 by zero
(ZeroDivide)

topaz 1> where
==> 1 ZeroDivide (AbstractException) >> _signalWith: @5 line 25
2 ZeroDivide (AbstractException) >> signal      @2 line 47
3 SmallInteger (Number) >> _errorDivideByZero   @6 line 7
4 SmallInteger >> /                             @6 line 7
5 Executed Code                                 @2 line 1
6 GsNMethod class >> _gsReturnToC              @1 line 11
```

select the section of the line after the frame number and before the step point, and use that as an argument to lookup:

```
topaz 1> lookup SmallInteger (Number) >> _errorDivideByZero
    category: 'Error Handling'
method: Number
_errorDivideByZero

"Generates a divide by 0 error."

^ ZeroDivide new _number: 2026 ; reason:
'numErrIntDivisionByZero';
    dividend: self ;
    signal
%
```

METHOD

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 as the first character in the line. 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 / OBJ2

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.

These commands cannot be abbreviated.

OBJ1Z / OBJ2Z

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.

These commands cannot be abbreviated.

OBJECT

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 38 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`.

Unordered collections (NSCs) 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: $l
   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

omit *aDisplayFeature*

The **display** and **omit** commands control the display of Gemstone Smalltalk objects and other features related to output.

The **display** command turns on these display attributes, and the **omit** command turns them off.;

For more details on these attributes and the default values, see the **display** command on page 84.

alloops

`omit alloops`

Disables the display of OOPs of classes along with class names in object display and of the oops of primitive specials, stopping the effect of **display alloops**.

bytes

`omit bytes`

When displaying byte-format objects, do not include the decimal or hexadecimal value of each byte.

classoops

`omit classoops`

Legacy; equivalent to **omit alloops**.

decimalbytes

`omit bytes`

When displaying byte-format objects, do not include the decimal or hexadecimal value of each byte.

deprecated

`omit deprecated`

For topaz commands such as **strings** and **senders**, which return lists of methods, **omit deprecated** causes those results to omit methods which contain sends of `deprecated`, `deprecated:`, or `deprecatedNotification:`.

errorcheck

`omit errorcheck`

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

flushoutput

`omit flushoutput`

displays the immediate flushing of topaz output files other than stdout.

lineeditor

`omit lineeditor`

Disables the use of the Topaz line editor, stopping the effect of **display lineeditor**. Always disabled on Windows.

names

`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.

oops

`omit oops`

Do not display OOP values with displayed results.

pauseonerror

`omit pauseonerror`

Disables pauses in Topaz execution after errors, stopping the effect of **`display pauseonerror`**.

pauseonwarning

`omit pauseonwarning`

Disables pause in Topaz execution after a compiler warning occurs.

pushonly

`omit pushonly`

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

resultcheck

`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.

singlecolumn

`omit singlecolumn`

Disables effect of **display singlecolumn**.

stacktemps

`omit stacktemps`

Disables effect of **display stacktemps**.

versionedclassnames

`omit versionedclassnames`

Disable the display of the class version for instances of classes that are not the last one in the classHistory of that class.

zerobased

`omit zerobased`

Shows offsets of instance variables as one-based when displaying objects.To show offsets as zero-based, use the **`display zerobased`** command.

OUTPUT

output [push | append | pushnew | pop] *aFileName* [only]

Controls where Topaz output is sent. Normally Topaz sends output to standard output (stdout): generally the topaz console. 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.

Output is written with UTF-8 encoding, regardless of the current state of **fileformat** (page 104).

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

`output` *aFileName*
`output push` *aFileName*

Sends output to the specified file, as well as echoing to stdout. 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 (`&`), or use **append** instead of **push**.

If you use **output** without a subsequent **push**, **pushnew**, **append**, or **pop**, then **push** is assumed and the following token is used as the filename.

The command **push** must be typed in full, it cannot be abbreviated.

`output append` *aFileName*

Sends output to the specified file, as well as echoing to stdout. 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*.

The command **append** must be typed in full, it cannot be abbreviated.

`output pushnew` *aFileName*

Sends output to the specified file, as well as echoing to stdout. 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 filenames of the form `foo.out`, the new filename will be `foo_N.out`, where where `N` is some integer between 1 and 99 (inclusive), and where `foo_N.out` did not previously exist. If more than 1000 versions of the file exist, the oldest version will be overwritten.

The command **push** must be typed in full, it cannot be abbreviated.

only

The above output commands will send output to both stdout and the each file on the stack. Using the **only** command both limits output to only go to the specific named file, and turns off the echo of results to stdout.

`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 stdout.

pop

`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.

The command **pop** must be typed in full, it cannot be abbreviated.

PAUSEFORDEBUG

pausefordebug [*errorNumber*]

Provided to assist internal debugging of a session.

With no argument, this command has no effect.

This command cannot be abbreviated.

PKGLOOKUP

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 136.

POLLFORSIGNAL

pollforsignal

Provided to assist debugging signal handling.

This command causes topaz to wait for out-of-band activity from the Gem processing, using GciPollForSignal. It can be interrupted by control-C.

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 `%` as the first character in the line. 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*.

Note that **printit** always displays results at level 1, regardless of the current display level setting (page 123). The **printit** command does not alter the current level setting. The **run** command (page 163) displays according to the current level setting, and the **doit** command (page 88) 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.

This command cannot be abbreviated.

QUIT

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.

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. Only 8 bits of the integer are returned. To get valid return values, the argument should always resolve to a value in the range 0..255. To use an object specification, including errorcount, you must be logged in when the **quit** command is executed.

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).

quit is ignored when reading a file using **input**, when stdin is a tty. If topaz stdin is redirected to a file and the file does not end with a **quit** or **exit**, topaz considers EOF on stdin to be an error, and will result in a non-zero topaz exit status.

This command cannot be abbreviated.

The command has the same behavior as **exit**. for more information and examples, see **exit** on page 96.

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 *GemBuilder for C* Manual. **releaseall** is equivalent to the GemBuilder for C call **GciReleaseOops**.

REMARK

remark *commentText*

Begins a remark (comment) line. Topaz ignores all succeeding characters on the line.

You can also use an exclamation point (`!`) or pound sign (#) as the first character in the line to signal the beginning of a comment.

```
topaz 1>      remark this is a comment
topaz 1> ! another comment
topaz 1> #       and yet another one
```

Comments are important in annotating Topaz batch processing files, such as test scripts.

REMOVEALLCLASSMETHODS

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.

This command removes all methods in all compilation environments, not just **env 0**.

This command cannot be abbreviated.

REMOVEALLMETHODS

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.

This command removes all methods in all compilation environments, not just **env 0**.

This command cannot be abbreviated.

RESUME

resume

In a topaz session created by **debuggem**, executes `System cancelWaitForDebug` in the Gem or topaz -l being debugged and then does a GciLogout from that gem. The GsProcess being debugged will be continued after the logout and `System class >> waitForDebug` will return to the caller.

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 `%` as the first character in the line. For example:

```
topaz 1> printit
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*.

The **run** command is similar to **printit**, with one significant difference. The **run** command uses the current display level setting (page 123), whereas **printit** always displays the result as if level 1 were the most recent **level** command.

RUNBLOCK

The topaz command **runblock** is primarily intended to support internal debugging.

runblock takes two arguments on the command line, and the following lines up to the next % must be the source for a block with between 0 and 10 block variables. The arguments are both objects per object specification format, see "Specifying Objects" on page 38.

The first argument is used for self in the block, and can be anything that can be specified on the command line.

The second argument must specify an Array of size N, N <= 10, where N is the number of argument variables. This second argument can be specified using ** or @OOP, or named in UserGlobals.

Example

```
topaz 1> run
  { 5 . 66 }.
%
a Array
  #1 5
  #2 66
topaz 1> runblock 'abc' **
  [:a :b |  self copy , a asString , b asString ]
%
abc566
```

RUNENV *envId*

Similar to run, but takes one integer argument >= 0 and <= 255, specifying compilation environment for the code to execute in. The argument takes precedence over the value set by **env** or **set compile_env**.

If GEM_NATIVE_CODE_ENABLED=FALSE in the gem configuration file, or if any breakpoints are set, execution defaults to interpreted mode, otherwise execution defaults to using native code.

SEND

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 38) 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*.

SENDERS

senders *selectorSpec*

Displays a list of all classes that are senders of the given *selectorSpec* (either a String or a Symbol). For example:

```
topaz 1> senders asByteArray
ByteArray >> copyReplaceAll:with:
ByteArray >> copyReplaceFrom:to:with:
```

This command is equivalent to the following

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

This command may use significant temporary object memory. Depending on your repository, 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*.

SET

set *aTopazParameter* [*aParamValue*]

The **set** command allows you to set session-specific values for your topaz session. This includes the GemStone login parameters, and settings that affect your topaz user interface.

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
```

cacert

`set cacert:` *aCaCertFilePath*

Sets the path to the trusted X509 certificate authority (CA) certificate to be used to validate certificates presented by peers. The certificate must be in PEM format.

This setting is used only by X509 logins and is cleared if traditional login parameters are set (**username**, **password**, **hostusername**, **hostpassword**, **gemstone**, **gemnetid**, or **sologin**).

cachename

`set cachename:` *aString*

Sets the current cachename, which applies to subsequent logins for all sessions. This cache name is recorded in the cache statistics collected by statmonitor for viewing in VSD. The name must be 31 characters or less, and is truncated if too long. The cachename can also be set using the command line **-u** argument. The colon is not required.

Using **set cachename** or the **-u** option has advantages over the programmatic assignment using `System class >> cacheName:`. 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.

category

`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 specified category does not 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.

cert

`set cert:` *aCertFilePath*

Sets the path to the X509 certificate to be used for login. The certificate must be in PEM format. The file may contain multiple certificates which comprise a certificate chain. The user certificate must be the first certificate in the file and may be followed by certificate authority (CA) certificates. All certificates must be ordered within the file such that the a given certificate is signed by the certificate which follows it.

This setting is used only by X509 logins and is cleared if traditional login parameters are set (**username**, **password**, **hostusername**, **hostpassword**, **gemstone**, **gemnetid**, or **sologin**).

class

`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, or by executing **set class** nil.

To display the name of the current class, issue the **set class** command without a class name.

compile_env

`set compile_env:` *anInteger*

Sets the compilation environmentId used for method compiliations and **run**, **printit**, etc. *anInteger* must be between 0 and 255 and is 0 by default. The compilation environment may also be set using the commend **env**.

directory

`set directory:` *directoryPath*

Set the name of the working directory for the RPC gem created by an X509 login.

This setting is used only by X509 logins and is cleared if traditional login parameters are set (**username**, **password**, **hostusername**, **hostpassword**, **gemstone**, **gemnetid**, or **sologin**).

editorname

`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.

enableremoveall

`set enableremoveall` *onOrOff*

When **set enableremoveall** is set to ON, it enables the commands **removeallmethods** and **removeallclassmethods**. When it set to OFF, these commands have no effect. The default is ON.

envvar

`set envvar:` *varName varValue*

Set an enviroment variable value in the topaz process. For example,

```
topaz 1 > set envvar MyRootDir /users/gdmin/project
```

The word **envvar** cannot be abbreviated.

extragemargs

`set extragemargs:` *aStringOfArgs*

Sets a list of extra command line arguments to be used when starting an RPC gem for an X509 login.

This setting is used only by X509 logins and is cleared if traditional login parameters are set (**username**, **password**, **hostusername**, **hostpassword**, **gemstone**, or **gemnetid**).

gemnetid

`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`. 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** or **-l)**, the default gemnetid is `gcilnkobj`. Use the status command to verify that this parameter is `gcilnkobj`. This makes the first session to log in a linked session. It is only possible to have one linked session per topaz process.

This command, like other set options, can be used in a `.topazini` initialization file that is executed when topaz starts up. When using **topaz -L**, any option to gemnetid within a **.topazini** file is ignored

When using linked topaz, if **gemnetid** is explicitly set to a gem service such as `gemnetobject`, login starts RPC sessions. Note that when you login an RPC session that way, the configuration parameters reported for the linked session do not apply; the rules governing configuration parameters for RPC sessions take effect.

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`. See the *System Administration Guide* for more on NRS syntax and usage.

This setting is used only by traditional logins and is cleared if any X509 login parameters are set. (**cert**, **cacert**, **key**, **netldi**, **logfile**, **directory**, or **extragemargs**)

gemstone

`set gemstone:` *aGemStoneName*

Specifies the name of the GemStone you want to log in to, in NRS syntax.

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
```

If the Stone is running on the same node as the Gem, the *node* portion of the string is optional; you only need to specify the Stone name: `gs64stone`. See the *System Administration Guide* for more on NRS syntax and usage.

This setting is used by traditional logins and is cleared if any X509 login parameters are set. (**cert**, **cacert**, **key**, **netldi**, **logfile**, **directory**, or **extragemargs**)

history

`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.

hostpassword

`set hostpassword:` *aPassword*

Sets the host password to be used when you next log in. If you don't include the host password argument 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.

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.

This setting is used only by traditional logins and is cleared if any X509 login parameters are set. (**cert**, **cacert**, **key**, **netldi**, **logfile**, **directory**, or **extragemargs**)

hostusername

`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.

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.)

This setting is used only by traditional logins and is cleared if any X509 login parameters are set. (**cert**, **cacert**, **key**, **netldi**, **logfile**, **directory**, or **extragemargs**)

inconversion

`set inconversion`

For use only by repository upgrade scripts in $GEMSTONE/upgrade. Sets the interal variable GciSupDbInConversion to TRUE.

inputpauseonerror

`set inputpauseonerror:` *onOrOff*

The argument must be a case-insensitive string, ON or OFF. If ON, and stdin is a tty, then the first **input** command issued interactively will transition topaz to **display pauseonerror**. This remains in effect until topaz input returns to stdin, at which point the previous state of **display pauseonerror** is restored.

The default is OFF

May not be abbreviated.

key

`set key:` *privateKeyPath*

For an X509 login, sets the path to the private key for the certificate specified by the SET CERT: command. The key must be in PEM format and must not be protected by a passphrase.

This setting is used only by X509 logins and is cleared if traditional login parameters are set (**username**, **password**, **hostusername**, **hostpassword**, **gemstone**, **gemnetid**, or **solologin**).

limit

`set limit:` *anInt*

Sets the limit on the number of bytes to display. The equivalent of **limit bytes** *anInt*

listwindow

`set listwindow:` *anInt*

Defines the maximum number of source lines to be listed by the **listw / l** command (page 130).

logfile

`set logfile:` *pathAndFilename*

For an X509 login, sets the name of the RPC gem's log file.

This setting is used only by X509 logins and is cleared if traditional login parameters are set (**username**, **password**, **hostusername**, **hostpassword**, **gemstone**, **gemnetid**, or **sologin**).

onetimepassword

`set onetimepassword` *onOrOff*

Determines if the password supplied via set password (or queried for interactively) is a normal GemStone password or a special one-time password. When OFF, the default, the password field is interpreted as a normal GemStone password.When ON, the password field is interpreted as a onetime password.

A one-time password is a temporary password valid for a single login for a specific user ID within a specific timeout. See the *System Administration Guide* for more details.

netldi

`set netldi:` *hostOrIp:portOrServiceName*

Sets the host and port for the netldi to be used for X509 certificate login. The argument format must include the hostName or the IP of the NetLDI host, and the listening port number of NetLDI service name, separated by a colon character. For example,

```
set netldi devhost.acme.com:54321
```

This setting is used only by X509 logins and is cleared if traditional login parameters are set (**username**, **password**, **hostusername**, **hostpassword**, **gemstone**, or **gemnetid**).

nrsdefaults

`set nrsdefaults:` *aNRSheader*

Sets the default components to be used in network resource string specifications. The parameter *aNRSheader* 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.

password

`set password:` *aGemStonePassword*

Sets the GemStone password to be used when you next log in. If you don't include the password argument 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.

This setting is used only by traditional logins and is cleared if any X509 login parameters are set. (**cert**, **cacert**, **key**, **netldi**, **logfile**, **directory**, or **extragemargs**)

session

`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.

sessioninit

`set sessionInit` *ONorOFF*

Provided to allow the execution of `GsCurrentSession >> initialize` to be skipped during login, in the case of upgrade or other special cases in which code invoked by GsCurrentSession cannot be executed and logins cannot complete. This requies that the Stone configuration parameter STN_ALLOW_NO_SESSION_INIT be enabled, which can be done at runtime by SystemUser.

This should be left at the default, ON, except for such exceptional cases.

singlecolumn

`set singlecolumn:` *onOrOff*

When stacks are printed, methods that have more than eight method arguments and temporary variables print their variables in four columns. The default is off. After executing **`set sologin: on`**, each variable will be displayed on a single line.

sologin

`set sologin:` *onOrOff*

The argument must be a case-insensitive string, ON or OFF. When ON, subsequent logins will be Solo, using the extent0.dbf specified by the GEM_SOLO_EXTENT config item. The default is OFF.

Setting solo login to on clears the X509 parameters; only traditional logins with **set user** and **set password** settings are legal for solo login.

May be abbreviated as SOLO.

sourcestringclass

`set sourcestringclass:` *ClassRangeSpecifier*

Sets the class of strings used to instantiate Smalltalk source strings generated by the **run**, **printit**, **doit**, **edit**, **method**, and **classmethod** commands. This includes any literal strings in the evaluated code.

This command expects one argument, which must be String orUnicode16. The options are:

`set sourcestringclass String`
: New instances of literal strings are created as instances of String, DoubleByteString, or QuadByteString.

`set sourcestringclass Unicode16`
: New instances of literal strings are created as instances of Unicode7, Unicode16, or Unicode32.

The Topaz **status** command shows the current setting.

On topaz startup, **sourcestringclass** is set to String. On login, the setting will be updated from the setting for #StringConfiguration in the GemStone Globals

SymbolDictionary. If #StringConfiguration resolves to Unicode16, then **sourcestringclass** will be set to Unicode16.

To avoid misinterpretation of fileouts, the **fileout** command writes a set sourcestringclass command at the start of the fileout. A **set sourcestringclass** command within a file only has effect within that file and any nested files.

stackpad

`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)

stackpad cannot be abbreviated.

tab

`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)

transactionmode

`set transactionmode` *aMode*

Set the current session's transaction mode, and set the transaction mode to this mode after each subsequent login. Must be one of (case-insensitive) autoBegin, manualBegin, or transactionless.

This command does an abort. If in a transaction, any uncommitted changes in the transaction will be lost. If the new mode is autoBegin, then a new transaction will be started.

username

`set username:` *aGemStoneUsername*

Establishes a GemStone user ID for the next login attempt.

This setting is used only by traditional logins and is cleared if any X509 login parameters are set. (**cert**, **cacert**, **key**, **netldi**, **logfile**, **directory**, or **extragemargs**)

SHELL

shell [*aHostCommand*]

spawn [*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).

For example, on Windows:

```
topaz 2> shell
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\GS64\32> dir *.txt
 Volume in drive C is Windows7_OS
 Volume Serial Number is 9ECC-468B

 Directory of C:\GS64\32

02/11/2014  04:38 PM            54,298 open_source_licenses.txt
02/11/2014  04:38 PM             3,209 PACKING.txt
02/11/2014  04:38 PM               104 version.txt
               3 File(s)         57,611 bytes
               0 Dir(s)  135,272,591,360 bytes free

C:\GS64\32>exit
```

```
topaz 2>
```

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).

If you supply parameters on the **shell** command line, they pass to a subprocess as a command for execution, and the output of the command is shown.

For example:

```
topaz 1> shell startnetldi -v
startnetldi 3.7.0 COMMIT: 2023-07-17T17:31:08-07:00
925e19317f5ac852dd751da32230481ad24c31a7
```

When issued with parameters, **shell** always creates a shell of the system default type (either Bourne or Korn).

spawn is the same as shell, and is included for compatibility with previous versions.

STACK

stack [*aSubCommand*]

Topaz can maintain up to 500 simultaneous GemStone Smalltalk process 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 examine and manipulate the contexts in the active call stack.

Debugging usually proceeds on the active call stack, but you may also save the active call stack before executing other code, and return to it later.

This command cannot be abbreviated.

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 GsNMethod
- 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/alloops** is active)
- the OOP of the GsNMethod (if **display oops/alloops** is active)

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

Any further commands that execute GemStone Smalltalk code: **run, printit**, **send**, **doit**, **step**, **edit last**, or **edit new text**, discards the active call stack unless **stack save** is executed.

`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.

Example 4.1 Example of stack display

```
topaz 1> run
{ 1 . 2 } do: [:x | x / 0 ]
%
ERROR 2026 , a ZeroDivide occurred (error 2026),
reason:numErrIntDivisionByZero, attempt to divide 1 by zero
(ZeroDivide)

topaz 1> stack
==> 1 ZeroDivide (AbstractException) >> _signalWith: @5 line 25
    receiver a ZeroDivide occurred (error 2026),
reason:numErrIntDivisionByZero, attempt to divide 1 by zero
    handleInCextensionBool nil
    res nil
(skipped 1 evaluationTemps)
2 ZeroDivide (AbstractException) >> signal      @2 line 47
    receiver a ZeroDivide occurred (error 2026),
reason:numErrIntDivisionByZero, attempt to divide 1 by zero
3 SmallInteger (Number) >> _errorDivideByZero   @6 line 7
    receiver 1
4 SmallInteger >> /                             @6 line 7
    receiver 1
    aNumber 0
5 [] in  Executed Code                          @2 line 1
    self nil
    receiver anExecBlock1
    x 1
6 Array (Collection) >> do:                     @5 line 10
    receiver anArray
    aBlock anExecBlock1
    i 1
(skipped 4 evaluationTemps)
7 Executed Code                                 @2 line 1
    receiver nil
8 GsNMethod class >> _gsReturnToC              @1 line 11
    receiver nil
```

Display or Redefine the Active Context

`stack scope`

Displays the current context (Scope is an alternate older name for context or frame). For example:

```
topaz 1> stack scope
1 AbstractException >> _signalWith:  @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 current 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.

`stack down`

Moves the current context down one level away from the top of the stack and displays the new context.

`stack set` *aGsProcess*

The argument is an object specification that resolves to a GsProcess. Make that aGsProcess the currently active stack for debugging.

`stack terminate`

Sends #terminate to the GsProcess of the current stack, if the stack is not owned by the scheduler.

`stack trim`

Trims the stack so that the current context 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 context 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 the Active Call Stack During Further Execution

When you have an active call stack, and execute any of the commands **run**, **printit**, **send**, **doit**, **edit last**, or **edit new text**, it results in the current call stack being discarded.

`stack save`

Save the active call stack before executing any of the commands that normally clear the stack:.

`stack nosave`

Cancel the previous **stack save.**

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
      0:  1 Animal >> habitat                   @1 line 1
      1:  1 AbstractException >> _signalWith:   @6 line 25
      *2:  1 Executed Code                       @3 line 1
```

The * indicates the active call stack, if one exists. If there are no saved stacks, a message to that effect is displayed.

Equivalent to **`threads`** (page 192)

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 the top context of the newly selected call stack.

Equivalent to **`thread`** *anInt* (page 191).

For example:

```
topaz 1> stack all
 0:  1 Animal >> habitat                    @1 line 1
 1:  1 AbstractException >> _signalWith:    @6 line 25
*2:  1 Executed Code                        @3 line 1
topaz 1> stack change 1
Stack 1 , GsProcess 27447553
1 AbstractException >> _signalWith:         @6 line 25
topaz 1> stack all
 0:  1 Animal >> habitat                    @1 line 1
*1:  1 AbstractException >> _signalWith:    @6 line 25
 2:  1 Executed Code                        @3 line 1
```

`stack set` *aGsProcessSpecification*

The argument is an object specification that resolves to a GsProcess. This GsProcess is made the active call stack for debugging.

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**.

Equivalent to **`thread`** *anInt* **`clear`** (page 191)

`stack delete all`

Removes all call stacks.

Terminate the process

`stack terminate`

Sends #terminate to the GsProcess of the current stack, if the stack is not owned by the scheduler.

STATUS

Displays your current login settings and other information about your Topaz session. These settings are set to default values when Topaz starts, and may be modified using the **set** (page 168) command, **display** (page 84) and **omit** (page 147), and using individual commands such as **level** (page 123) , **limit** (page 124), and **fileformat** (page 104).

For example:

```
topaz 1> status

Current settings are:
 display level: 0
 byte limit: 0 lev1bytes: 100
 omit bytes
 include deprecated methods in lists of methods
 display instance variable names
 display oops   omit alloops   omit stacktemps
 oop limit: 0
 omit automatic result checks
 omit interactive pause on errors
 omit interactive pause on warnings
 listwindow: 20
 stackpad: 45 singlecolumn: Off  tab (ctl-H) equals 8 spaces when
listing method source
 transactionmode  autoBegin
 using line editor
   line editor history: 100
   topaz input is from a tty on stdin
EditorName_________ vi
CompilationEnv____ 0
Source String Class String
fileformat          utf8
SessionInit         On
EnableRemoveAll     On
CacheName_________ 'TopazR'

Connection Information:
UserName___________ 'Isaac_Newton'
Password __________ (set)
HostUserName_______ 'newtoni'
HostPassword_______ (set)
NRSdefaults________ '#netldi:gs64ldi'
GemStone___________ 'gs64stone'
GemStone NRS_______ '!#encrypted:newtoni@password#server!gs64stone'
GemNetId___________ 'gemnetobject'
GemNetId NRS_______ '!#encrypted:newtoni@password!gemnetobject'

Browsing Information:
Class_____________
Category__________ (as yet unclassified)
```

STEP

step (over | into | thru)

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 3, "Debugging Your GemStone Smalltalk Code", starting on page 55.

`step`

Equivalent to **step over**.

`step over`

Advances execution to the next step point in the current frame or its caller. The current frame is the top of the stack or the frame 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.

`step thru`

Advances execution to the next step point in the current frame, or its caller, or the next step point in a block for which current frame's method is the home method.

STK

stk [*aSubCommand*]

Similar to **stack**, but does not display parameters and temporaries for each frame. All frames for the active call stack are displayed, with the current active frame indicated by an arrow.

For more information on s, see the **stack** command on page 177.

This command cannot be abbreviated.

```
topaz 1> printit
{ 1 . 2} do: [:x | x / 0 ]
%
ERROR 2026 , a ZeroDivide occurred (error 2026),
reason:numErrIntDivisionByZero, An attempt was made to divide 1 by
zero. (ZeroDivide)
topaz 1> stk
==> 1 ZeroDivide (AbstractException) >> _signalWith: @6 line 25
2 ZeroDivide (AbstractException) >> signal      @2 line 47
3 SmallInteger (Number) >> _errorDivideByZero   @6 line 7
4 SmallInteger >> /                             @6 line 7
5 [] in  Executed Code                          @2 line 1
6 Array (Collection) >> do:                     @5 line 10
7 Executed Code                                 @2 line 1
8 GsNMethod class >> _gsReturnToC              @1 line 11
```

STRINGS

strings *selectorSpec*

Displays a list of all methods that contain the given *selectorSpec* (either a String or a Symbol) in their source string, and class comments in which the text includes *selectorSpec*.

Search is case-sensitive; for a case-insensitive search, see **stringsic**. This command cannot be abbreviated.

For example:

```
topaz 1> strings ChangeUserId
UserProfile >> privileges
UserProfile >> userId:password:
UserProfile >> _privileges
UserProfile class >> _initPrivilegeNames
UserProfileSet >> _oldUserId:newUserId:for:
```

The **strings** command is equivalent to the following:

```
topaz 1> run
| org |
org := ClassOrganizer new.
org stringsReport: aString ignoreCase: false
   includeClassComments: true
%
```

This command may use significant temporary object memory. Depending on your repository, 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*.

STRINGSIC

stringsic *selectorSpec*

Displays a list of all methods that contain the given *selectorSpec* (either a String or a Symbol) in their source string, and class comments that include *selectorSpec*.

This search is case-insensitive; for a case-sensitive search, see the **strings** command. This command cannot be abbreviated.

The **stringsic** command is equivalent to the following:

```
topaz 1> run
| org | org := ClassOrganizer new.
org stringsReport: 'referencesToLiteral:' ignoreCase: true
   includeClassComments: true
%
```

This command may use significant temporary object memory. Depending on your repository, 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*.

SUBCLASSES

subclasses [*aClassName*]

Prints immediate subclasses of the specified class. If you don't specify a class name, prints subclasses of the current class.

```
topaz 1> subclasses MultiByteString
DoubleByteString
QuadByteString

topaz 1> set class DoubleByteString
topaz 1> subclasses
DoubleByteSymbol
Unicode16
```

For command to print the complete hierarchy, see **subhierarchy** on page 187.

SUBHIERARCHY

subhierarchy [*aClassName*]

Print print a hierarchy report for all subclasses of the specified class. If you don't specify a class name, prints hierarchy report for the current class.

Example

```
topaz 1> subhierarchy MultiByteString
MultiByteString
  DoubleByteString
    DoubleByteSymbol
    Unicode16
  QuadByteString
    QuadByteSymbol
    Unicode32
```

See Also

hierarchy (page 111)

TEMPORARY

temporary [*aTempName*[/*anInt*] [*anObjectSpec*]]

Displays or redefines the value of one or more temporary variables in the current frame of the current stack. For examples and other useful information, see Chapter 3, "Debugging Your GemStone Smalltalk Code", starting on page 55.

All Topaz object specification formats (as described in "Specifying Objects" on page 38) are legal in **temporary** commands.

temporary
Displays the names and values of all temporary objects in the current frame.

temporary *aTempName*
Displays the value of the first temporary object with the specified name in the current frame.

```
topaz 1> temporary preferences
preferences       an Array
```

temporary *aTempName anObjectSpec*
Redefines the specified temporary in the current frame to have the value *anObjectSpec*.

temporary *anInt*
Displays the value of the temporary at offset *n* in the current frame. 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 current frame to have the value *anObjectSpec*.

For example, to view the temporary variable values:

```
topaz 1> break classmethod String withAll:
topaz 1> run
String withAll: 'abc'
%
a Breakpoint occurred (error 6005), Method breakpoint encountered.
1 String class >> withAll:        @1 line 1
topaz 1> stack
==> 1 String class >> withAll:                      @1 line 1
    receiver String
    aString abc
2 Executed Code                                   @2 line 1
    receiver nil
3 GsNMethod class >> _gsReturnToC              @1 line 11
    receiver nil
```

and to modify the value of the temporary:

```
topaz 1> temporary aString 'xyz'
topaz 1> stack
==> 1 String class >> withAll:                      @1 line 1
    receiver String
    aString xyz
2 Executed Code                                   @2 line 1
    receiver nil
3 GsNMethod class >> _gsReturnToC                 @1 line 11
    receiver nil
```

the method will return the modified value:

```
topaz 1> continue
xyz
```

When the Topaz command **display oops** has been set, temporaries displayed as `.tN` are un-named temporaries private to the virtual machine. The example below displays the temporaries used in evaluation of the optimized to:do:, both as shown by the **frame** command and by the **temporary** command.

```
topaz 1> run
| a |
1 to: 25 do: [:j | a := j. a pause]
%
...
topaz 1> display oops
topaz 1> frame 5
5 Executed Code                      @4 line 2   [methId 25464833]
    receiver [20 sz:0 cls: 76289 UndefinedObject] nil
    a [10 sz:0 cls: 74241 SmallInteger] 1 == 0x1
    j [10 sz:0 cls: 74241 SmallInteger] 1 == 0x1
    .t1 [10 sz:0 cls: 74241 SmallInteger] 1 == 0x1
    .t2 [202 sz:0 cls: 74241 SmallInteger] 25 == 0x19
    .t3 [10 sz:0 cls: 74241 SmallInteger] 1 == 0x1
    .t4 [10 sz:0 cls: 74241 SmallInteger] 1 == 0x1
topaz 1> temporary
    a [10 sz:0 cls: 74241 SmallInteger] 1 == 0x1
    j [10 sz:0 cls: 74241 SmallInteger] 1 == 0x1
    .t1 [10 sz:0 cls: 74241 SmallInteger] 1 == 0x1
    .t2 [202 sz:0 cls: 74241 SmallInteger] 25 == 0x19
    .t3 [10 sz:0 cls: 74241 SmallInteger] 1 == 0x1
    .t4 [10 sz:0 cls: 74241 SmallInteger] 1 == 0x1
```

TFILE

tfile *aFile*

Input a tonel format class definition or class extension file. The methods within the file are all compiled.

The Class specified in the file must already exist (i.e. class definition execution is not attempted).

THREAD

thread [*anInt*] [clear]

Displays the currently selected GemStone process from among the stack saved from the last error, or from those retrieved by the most recent **threads** command.

```
topaz 1> thread
Stack 0 , GsProcess 27462401
1 Animal >> habitat                         @1 line 1
```

thread *anInt*

Changes the currently selected GemStone process. You can specify an integer value from among those shown in the most recent **threads** command.

```
topaz 1> thread 1
Stack 1 , GsProcess 27447553
1 AbstractException >> _signalWith:         @6 line 25
```

thread *anInt* **clear**

Clears the selected GsProcess from the Topaz stack cache.

THREADS

threads [clear]

Force any dirty instances of GsProcess cached in VM stack memory to be flushed to object memory. It then executes a message send of

`ProcessorScheduler >> topazAllProcesses`

and retrieves and displays the list of processes.

```
topaz 1> threads
      0: 27462401 debug
=> 1: 27447553 debug (topaz current)
      2: 27444225 debug
```

`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.

Example

```
topaz 1> time
02/06/2023 13:37:13.545 PST
CPU time:   0.035 seconds
topaz 1> printit
Array allInstances size
%
23515
topaz 1> time
02/06/2023 13:37:48.459 PST
CPU time:   0.232 seconds
Elapsed Real time:   8.649 seconds
Elapsed CPU  time:   0.083 seconds
```

TMETHOD

Compile a method from tonel format used by the Rowan open source project. This is intended for interactive use.

```
topaz 1 > tmethod
{ #category : 'test' }
TestClass >> version[
      ^5
]
```

This compiles the method using the class from the tonel method definition (in the example, TestClass), and using the current compilation environment. It does not affect topaz's current class.

The closing] must be the first character on a line.

TOPAZWAITFORDEBUG

Waits forever in a sleep loop until a debugger is attached to continue execution.

This can be a C debugger (gdb, etc.), or (if configured appropriately), another topaz session attaching via **debuggem**.

This command should be used with caution; it may require an OS-level kill to terminate the process. Intended for use in debugging; see "Debugging from a different session" on page 62 and the **debuggem** (page 78) and **iferr** (page 113) commands.

UNPROTECTMETHODS

Cancels the effect of **protectmethods**, which is used for consistency checking in filein scripts.

This command cannot be abbreviated.

UP

up [*anInteger*]

In the current stack, change the current frame to be the caller of the current frame, and display the new selected frame. The optional argument *anInteger* specifies how many frames to move up. If no argument is supplied, the scope will go up one frame.

The behavior is similar to **stack up**, except that stack up does not accept an argument, and the frame display for stack up does not includes parameters and temporaries for the frame. **stack up** is described on page 179.

```
topaz 1> run
{ 1 . 2 } do: [:x | x / 0 ]
%
ERROR 2026 , a ZeroDivide occurred (error 2026),
reason:numErrIntDivisionByZero, attempt to divide 1 by zero
(ZeroDivide)
topaz 1> where
==> 1 ZeroDivide (AbstractException) >> _signalWith: @5 line 25
2 ZeroDivide (AbstractException) >> signal      @2 line 47
3 SmallInteger (Number) >> _errorDivideByZero   @6 line 7
4 SmallInteger >> /                             @6 line 7
5 [] in  Executed Code                          @2 line 1
6 Array (Collection) >> do:                     @5 line 10
7 Executed Code                                 @2 line 1
8 GsNMethod class >> _gsReturnToC              @1 line 11

topaz 1> up 4
5 [] in  Executed Code                          @2 line 1
    self nil
    receiver anExecBlock1
    x 1

topaz 1> where
1 ZeroDivide (AbstractException) >> _signalWith: @5 line 25
2 ZeroDivide (AbstractException) >> signal      @2 line 47
3 SmallInteger (Number) >> _errorDivideByZero   @6 line 7
4 SmallInteger >> /                             @6 line 7
==> 5 [] in  Executed Code                          @2 line 1
6 Array (Collection) >> do:                     @5 line 10
7 Executed Code                                 @2 line 1
8 GsNMethod class >> _gsReturnToC              @1 line 11
```

WHERE

where [*anInteger* | *aString*]

Displays the current call stack, with one line per frame.

where
Displays all lines of the current call stack. Equivalent to the **stk** command.

```
topaz 1> where
==> 1 ZeroDivide (AbstractException) >> _signalWith: @6 line 25
2 ZeroDivide (AbstractException) >> signal      @2 line 47
3 SmallInteger (Number) >> _errorDivideByZero   @6 line 7
4 SmallInteger >> /                             @6 line 7
5 Executed Code                                 @2 line 1
6 GsNMethod class >> _gsReturnToC              @1 line 11
```

where *anInteger*
Displays the specified number of frames of the stack, starting with the current frame.

```
topaz 1> where 3
==> 1 ZeroDivide (AbstractException) >> _signalWith: @6 line 25
2 ZeroDivide (AbstractException) >> signal      @2 line 47
3 SmallInteger (Number) >> _errorDivideByZero   @6 line 7
```

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:

```
topaz 1> where error
==> 3 SmallInteger (Number) >> _errorDivideByZero   @6 line 7
```

Appendix

A

Topaz Command-Line Syntax

This section presents the formal command syntax and the available command-line options.

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]** [**-q**] [**-i** | **-I** *topazini*] [**-S** *scriptFile*] [**-n** *hostName*:*netldiName*] [**-u** *useName*] [**-X** *caCertPaths*] [**--** *other args*]

When invoked with the **-L** or **-l** option, Topaz runs in linked mode. The command line accepts additional options that apply only when starting linked version:

> **topaz -l | -L** [**-q**] [**-i** | **-I** *topazini*] [**-S** *scriptFile*] [**-u** *useName*]
> [**-e** *exeConfig* | **-E** *exeConfigFile*] [**-z** *systemConfig*] [**-T** *tocSizeKB*]
> [**-C** *configParams*] [**--** *other args*]

In linked topaz (topaz -l or -L), you may also login RPC sessions, as well as login a single linked session. Settings that only apply to linked sessions (**-e**, **-E**, **-z**, **-T**, and **-C**) do not apply for RPC sessions started from linked topaz.

topaz also provides usage and version information:

> **topaz -h | -v**

Arguments:

-C *configParams*	Provides configuration parameters, in configuration file syntax, that override settings in the configuration files. Only applies to linked sessions (RPC sessions may use the -C syntax in the Gem's NRS). This option may be included more than once; each string of **-C** argument parameters is applied in the order they appear. For example, `topaz -l -C 'GEM_TEMPOBJ_CACHE_SIZE = 1GB; GEM_TEMPOBJ_OOMSTATS_CSV = TRUE;'`

-e *exeConfig*	The GemStone executable configuration file or directory containing a file named gem.conf. This only applies to linked sessions. See the *System Administration Guide*, Appendix A, for details on configuration files.
-E *exeConfigFile*	The GemStone executable configuration file. Only one of **-e** and **-E** can be used. This only applies to linked sessions. See the *System Administration Guide*, Appendix A, for details on configuration files.
-h	Displays a usage line and exits.
-i	Ignore the initialization file, `.topazini.`
-I *topazini*	Specify a complete path and file to a topazini initialization files, and use this rather then any `.topazini` in the default location.
-l	Invoke the linked version of Topaz.
-L	Invoke the linked version of Topaz, and do not apply any command **set gemnetid** that may appear in the .topazini file or a file passed in using **-I**.
-n *hostName***:***netldiName*	For a login using X509-Secured GemStone only, to specify the NetLDI to spawn the Gem, part of the X509-secured GemStone login parameters. The host name or IP, and the netldi name or listening port, for an X509-secured NetLDI.
-q	Start Topaz in quiet mode, suppressing printout of the banner and other information.
-r	Invoke the RPC (remote procedure call) version of Topaz.
-S *scriptFile*	Specifies a script file that will be processed with INPUT.
-T *tocSizeKB*	The GEM_TEMPOBJ_CACHE_SIZE that will be used. Overrides any settings provided in configuration files passed as arguments with the **-e** or **-z** options. Only applies to linked sessions.
-u *useName*	Sets the cache name, as recorded by statmonitor for viewing in VSD. This is also useful for identifying processes in OS utilities such as `top` or `ps`.
-v	Prints version and exits.
-X *CaCertPaths*	For a login using X509-Secured GemStone only, to set certificates. Requires additional infrastructure to be running, including X509-secured NetLDIs and caches. The argument must specify three paths in the defined order: cacert, chained key for user, and private key for user. For example: `-X 'stoneCA-dev.cert.pem;DataCurator.chain.pem ;DataCurator.privkey.pem'`

-z *systemConfig*	The GemStone system configuration file or directory containing a specifically named file; if the **-E** argument is also used, only a file is accepted. This applies only to linked sessions. See the *System Administration Guide*, Appendix A, for details on configuration files.
-- *otherArgs*	Arbitrary text arguments *otherArgs* may be included after the "--" end of arguments marker, which must follow any of the above topaz arguments are included.