
GemStone®

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

Version 2.4

September 2009

GEMSTONE[™]S 64

INTELLECTUAL PROPERTY OWNERSHIP

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

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

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

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

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

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

COPYRIGHTS

This software product, its documentation, and its user interface © 1986-2009 GemStone Systems, Inc. All rights reserved by GemStone Systems, Inc.

PATENTS

GemStone is covered by U.S. Patent Number 6,256,637 "Transactional virtual machine architecture", Patent Number 6,360,219 "Object queues with concurrent updating", and Patent Number 6,567,905 "Generational Garbage Collector". GemStone may also be covered by one or more pending United States patent applications.

TRADEMARKS

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

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

Sun, **Sun Microsystems**, **Solaris**, and **SunOS** are trademarks or registered trademarks of Sun Microsystems, Inc. All **SPARC** trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. **SPARCstation** is licensed exclusively to Sun Microsystems, Inc. Products bearing **SPARC** trademarks are based upon an architecture developed by Sun Microsystems, Inc.

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

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

Microsoft, **MS**, **Windows**, **Windows 2000** and **Windows XP** are registered trademarks of Microsoft Corporation in the United States and other countries.

Linux is a registered trademark of Linus Torvalds and others.

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

AIX and **POWER4** are trademarks or registered trademarks of International Business Machines Corporation.

Other company or product names mentioned herein may be trademarks or registered trademarks of their respective owners. Trademark specifications are subject to change without notice. All terms mentioned in this documentation that are known to be trademarks or service marks have been appropriately capitalized to the best of our knowledge; however, GemStone cannot attest to the accuracy of all trademark information. Use of a term in this documentation should not be regarded as affecting the validity of any trademark or service mark.

GemStone Systems, Inc.

1260 NW Waterhouse Avenue, Suite 200
Beaverton, OR 97006.

About This Manual

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

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

Assumptions

To make use of the information in this manual, you must be familiar with the GemStone/S 64 Bit system and the GemStone Smalltalk programming language. In addition, you should be familiar with your host operating system. This manual assumes that the GemStone database system has been correctly installed on your host computer, using the instructions in the *GemStone/S 64 Bit Installation Guide*.

How This Manual Is Organized

- Chapter 1 introduces you to Topaz. You'll learn how to run Topaz, how to log in to the GemStone server, how to create and execute GemStone Smalltalk code, and how to inspect GemStone objects.
- Chapter 2 shows how to use Topaz to debug your GemStone Smalltalk code.
- Chapter 3 describes the Topaz commands in alphabetical order.
- Appendix A lists the Topaz command line syntax.
- Appendix B lists the syntax for specifying the host machine for a GemStone file or process.

Terminology Conventions

This document uses the following terminology:

- The term "GemStone" is used to refer both to the product, GemStone/S 64 Bit, or previous GemStone/S server products; and to the company, GemStone Systems, Inc.

Typographical Conventions

This document uses the following typographical conventions:

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

```
% env | grep GEM
```

- Topaz commands and UNIX commands and are shown in **bold** type. For example:

```
copyextent
```

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

```
markForCollection
```

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

```
StoneName.conf
```

Related GemStone Documentation

For more information about the GemStone/S 64 Bit database system and its development tools, refer to the following manuals:

- *Release Notes*. Describes information specific to a particular release of GemStone/S 64 Bit.
- *Installation Guide*. Describes installation and configuration of your GemStone/S 64 Bit system.
- *System Administration Guide for GemStone/S 64 Bit*. Describes maintenance and administration of your GemStone/S 64 Bit system.
- *Programming Guide for GemStone/S 64 Bit*. A programmer's guide to GemStone Smalltalk, GemStone's object-oriented programming language.

Technical Support

GemStone provides several sources for product information and support. The product-specific manuals provide extensive documentation, and should be your first source of information.

GemStone Web Site: <http://support.gemstone.com>

GemStone's Technical Support website provides a variety of resources to help you use GemStone products. Use of this site requires an account, but registration is free of charge. To get an account, just complete the Registration Form, found in the same location. You'll be able to access the site as soon as you submit the web form.

The following types of information are provided at this web site:

Documentation for GemStone/S 64 Bit is provided in PDF format. This is the same documentation that is included with your GemStone/S 64 Bit product.

Release Notes and **Install Guides** for your product software are provided in PDF format in the Documentation section.

Downloads and **Patches** provide code fixes and enhancements that have been developed after product release, and past and current versions of GemBuilder for Smalltalk.

Bugnotes, in the Learning Center section, identify performance issues or error conditions that you may encounter when using a GemStone product. A bugnote describes the cause of the condition, and, when possible, provides an alternative means of accomplishing the task. In addition, bugnotes identify whether or not a

fix is available, either by upgrading to another version of the product, or by applying a patch. Bugnotes are updated regularly.

TechTips, also in the Learning Center section, provide information and instructions for topics that usually relate to more effective or efficient use of GemStone products.

Community Links provide customer forums for discussion of GemStone product issues.

Technical information on the GemStone Web site is reviewed and updated regularly. We recommend that you check this site on a regular basis to obtain the latest technical information for GemStone products.

Help Requests

You may need to contact Technical Support directly for the following reasons:

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

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

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

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

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

For non-emergency requests, the support website is the preferred way to contact Technical Support. Only designated support contacts may submit help requests via the support website. If you are a designated support contact for your company, or the designated contacts have changed, please contact us to update the appropriate user accounts.

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

Email: support@gemstone.com

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

Requests for technical assistance may be submitted online, or by email or by telephone. We recommend you use telephone contact only for more serious requests that require immediate evaluation, such as a production system that is non-operational. In these cases, please also submit your request via the web or email, including pertinent details such error messages and relevant log files.

If you are reporting an emergency by telephone, select the option to transfer your call to the technical support administrator, who will take down your customer information and immediately contact an engineer.

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

24x7 Emergency Technical Support

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

—
|

Chapter 1. Getting Started with Topaz ***15***

1.1 Invoking Topaz	16
1.2 Overview of a GemStone Session	16
1.3 Remote Versus Linked Versions	17
1.4 Logging In to GemStone	18
Setting Up a Login Initialization File	21
1.5 The Help Command	23
1.6 Executing GemStone Smalltalk Expressions	24
1.7 Escaping to an Editor	25
1.8 Controlling the Display of Results	25
Display Level	26
Setting Limits on Object Displays	28
Displaying Variable Names, OOPs, and Hex Byte Values	28
Instance Variable Names	28
Hexadecimal Byte Values.	29
OOP Values	29
1.9 Creating and Changing Methods	30
Editing Methods.	31

1.10 Listing Methods and Categories.	32
1.11 Committing and Aborting Transactions	32
1.12 Capturing Your Topaz Session In a File	33
1.13 Filing Out Classes and Methods.	34
1.14 Creating a Topaz Script for Batch Processing	36
1.15 Taking Topaz Input from a File	37
1.16 Interrupting Topaz and GemStone	37
1.17 Multiple Concurrent GemStone Sessions.	38
1.18 Structural Access To Objects	39
Examining Instance Variables with Structural Access	40
Specifying Objects	41
Object Identity Specification Formats.	41
Literal Object Specification Formats.	41
1.19 Defining Local Variables	42
Creating Variables	43
Displaying Current Variable Definitions.	44
Clearing Variable Definitions	44
1.20 Sending Messages.	45
1.21 Logging Out	45
1.22 Leaving Topaz.	46

Chapter 2. Debugging Your GemStone Smalltalk Code **47**

2.1 Step Points and Breakpoints.	48
2.2 Setting, Clearing, and Examining Breakpoints	49
2.3 Examining the GemStone Smalltalk Call Stack	51
Proceeding After a Breakpoint	52
Examining and Modifying Temporaries and Arguments	53
Select a Context for Examination and Debugging.	54
Redefine the Active Call Stack.	54

Chapter 3. Command Dictionary **57**

ABORT.	59
BEGIN	60
BREAK aSubCommand	61

Method Breakpoints	61
Disabling and Enabling Breakpoints	62
CATEGORY: aCategoryName.	64
CLASSMETHOD[: aClassName]	65
COMMIT.	66
CONTINUE [anObjectSpec]	67
DEFINE [aVarName [anObjectSpec [aSelectorOrArg]...]]	68
DISASSEM [aClassParameter] aParamValue	69
DISPLAY aDisplayFeature.	70
DOIT	72
DOWN [anInteger]	73
EDIT aSubCommandOrSelector [aSelector]	74
Creating or Modifying Blocks of GemStone Smalltalk Code	74
Creating or Modifying GemStone Smalltalk Methods	74
ERRORCOUNT	76
EXIT [anInteger].	77
EXPECTBUG bugNumber	78
EXPECTERROR anErrorCategory anErrorNumber.	79
EXPECTVALUE anObjectSpec [anInt]	82
FILEOUT aSubCommandOrSelector [TOFILE: aFileName]	84
FRAME [anInteger]	85
GCITRACE aFileName.	86
HELP [aTopicName]	87
IFERR bufferNumber [aTopazCommandLine]	88
IFERR_CLEAR	89
IFERR_LIST	90
IFERROR [aTopazCommandLine]	91
INPUT [aFileName POP]	92
LEVEL anIntegerLevel	93
LIMIT [BYTES OOPS] anInteger	94
LIST.	95
Browsing Dictionaries and Classes	95
Listing Methods	95
Listing Step Points.	96
Listing Breakpoints	97
LOADUA aFileName.	98
LOGIN	99
LOGOUT.	101
LOOKUP.	102
Finding and Listing Methods	102
METHOD[: aClassName]	103

NBRUN	104
NBRESULT	105
OBJECT anObjectSpec [AT: anIndex [PUT: anObjectSpec]]	106
OMIT aDisplayFeature	108
OPAL	109
OUTPUT (PUSH APPEND PUSHNEW POP) aFileName [ONLY]	110
PRINTIT	112
PROTECTMETHODS	113
QUIT [anInteger]	114
RELEASEALL.	115
REMARK commentText	116
REMOVEALLMETHODS[: aClassName]	117
REMOVEALLCLASSMETHODS[: aClassName]	118
RUN	119
SEND anObjectSpec aMessage	120
SET aTopazParameter [aParamValue]	121
SHELL [aHostCommand]	126
SPAWN [aHostCommand]	127
STACK [aSubCommand]	128
Display the Active Call Stack	128
Display or Redefine the Active Context	130
Save or Delete the Active Call Stack During Execution.	131
Display All Call Stacks	131
Redefine the Active Call Stack	131
Remove Call Stacks	132
STATUS	133
STEP (OVER INTO)	134
STK [aSubCommand]	135
TEMPORARY [aTempName[/anInt] [anObjectSpec]]	136
TIME	138
UNPROTECTMETHODS	139
UP [anInteger]	140
WHERE [anInteger]	141

Appendix A. Topaz Command-Line Syntax ***143***

A.1 Command-Line Syntax	143
A.2 Options	144

Appendix B. Network Resource String Syntax ***147***

B.1 Overview 147
B.2 Defaults 148
B.3 Notation 149
B.4 Syntax 150

Index ***153***

—
|

Getting Started with Topaz

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

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

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

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

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

1.1 Invoking Topaz

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

Figure 1.1 Topaz Banner and Prompt

```
% topaz

GemStone/S64 Object-Oriented Data Management System
Copyright (C) GemStone Systems, Inc. 1986-2007.
All rights reserved.
-----
PROGRAM: topaz, Linear GemStone Interface (Remote Session)
VERSION: 2.4.0, Thu Aug 20 16:16:14 US/Pacific 2009
BUILD: gss64bit-15
BUILT FOR: SPARC (Solaris)
MODE: 64 bit
RUNNING ON: 2-CPU handel sun4u (Solaris 2.9 Generic_117171-08) 400MHz
sparcv9,2048MB
PROCESS ID: 3596      DATE: 08/21/09 15:41:54 PDT
USER IDS: REAL=gsuser (531) EFFECTIVE=gsuser (531)

topaz>
```

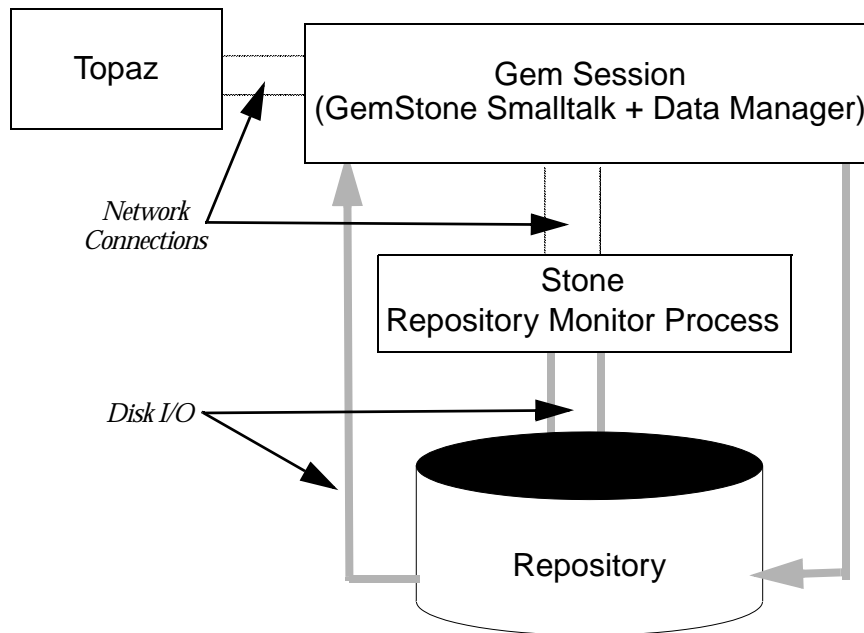
Type the **topaz** command and verify that it responds as shown. If you have problems invoking Topaz, review the requirements listed on the previous page or consult your system administrator or your GemStone data curator.

1.2 Overview of a GemStone Session

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

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

Figure 1.2 GemStone Object Server Components



1.3 Remote Versus Linked Versions

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

The `topaz -l` (for linked) command line invokes the linked version of Topaz. The linked version allows you to run multiple Topaz sessions, but session number one is always a linked session, where the Topaz session and a Gem exist as a single process. Any additional sessions are RPC. The linked Topaz session provides faster throughput than the RPC version, allows some administrative operations to report more complete details, and is required for certain system administration tasks, such as upgrading the repository.

Under Windows, only the RPC version of Topaz is available.

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

1.4 Logging In to GemStone

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

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

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

Otherwise, specify the name of the Stone. If the node where the Stone is running is not the one where the Gem will run, you also need the name of the Stone host and perhaps the type of network connection between the Stone and Gem hosts. To specify a process named `gs64stone` running on node `central`, you can use a network resource string of the form `!@central!gs64stone`. Your GemStone data curator can give you the exact string to use. Appendix B describes the syntax of network resource strings.

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

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

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

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

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

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

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

This is sufficient for the linked version of Topaz. If you are running the RPC version, you must also provide the following information:

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

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

```
topaz> status
```

```
Current settings are:
display level: 1
omit oops
omit bytes
display instance variable names
omit automatic result checks
omit interactive pause on errors
EditorName_____ vi

Connection Information:
UserName_____ 'Isaac Newton'
Password _____ (set)
HostUserName_____ 'newtoni'
HostPassword _____ (set)
GemStone_____ 'gs64stone'
GemStone NRS_____
'!#encrypted:newtoni@password#server!gs64stone'
GemNetId_____ 'gemnetobject'
GemNetId NRS_____
'!#encrypted:newtoni@password!gemnetobject'
CacheName_____ (default)
```

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

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

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

```
topaz> login
GemStone password? (type your GemStone password)
successful login
topaz 1>
```

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

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

```
topaz> set gemstone gs64stone user 'Isaac Newton'
topaz> set gemnetid gemnetobject hostuser 'newtoni'
topaz> set hostpass <return>
Host Password? (type your host password)
topaz> login
gci login: currSession 1 rpc gem processId 95
successful login
topaz 1>
```

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

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

```
topaz> set gemstone gs64stone user 'Isaac Newton' pass gravity
topaz> login
[Info]: LNK client/gem GCI levels = 830/830
[Info]: User ID: 'Isaac Newton'
[Info]: Repository: gs64stone
[Info]: Session ID: 5
[Info]: GCI Client Host: <Linked>
[Info]: Page server PID: -1
[Info]: Login Time: 08/21/09 16:03:17.187 PDT
gci login: currSession 1 rpc gem processId -1
successful login
topaz 1>
```

Setting Up a Login Initialization File

You can streamline the login process by creating an initialization file that contains the **set** commands needed for logging in. When you invoke Topaz, it automatically executes those commands for you. If you insert **set hostpassword** and **login**

commands without parameters, Topaz automatically prompts you for the necessary values.

Table 1.1 Topaz Initialization File Names

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

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

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

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

CAUTION:

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

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

Figure 1.3 Topaz Initialization File

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

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

Once Topaz has read an initialization file like this one, logging in is quite simple:

```
% topaz
```

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

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

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

1.5 The Help Command

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

```
topaz 1> help exit
```

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

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

If an nonzero integer status is given, a nonzero status will be returned to the operating system. A zero status will return a zero status to the operating system.

Help is available for:

(list of topics)

Topic? *(press Return to exit the help utility)*

```
topaz 1>
```

1.6 Executing GemStone Smalltalk Expressions

By following the examples in the rest of this chapter, you'll learn how to create and execute GemStone Smalltalk code, and how to inspect GemStone objects. If you need to log out of your session before you finish, you can use the **commit** command to save the classes and methods you have created. To start where you left off in a new session, you will have to reset the current class and category, but usually not the default screen display settings.

Once you've logged in to GemStone, you can execute Smalltalk expressions with the **printit** command. The following use of **printit**, for example, creates a class named `Animal`.

```
topaz 1> printit
Object subclass: 'Animal'
    instVarNames: #('name' 'favoriteFood' 'habitat')
    inDictionary: UserGlobals
%
Animal class
  superClass      Object class
  format          0
  instVars        3
  instVarNames    an Array
  constraints      an Array
  classVars       a SymbolDictionary
  methodDict      a GsMethodDictionary
  poolDictionaries an Array
  categories      a GsMethodDictionary
  secondarySuperclasses nil
  name            Animal
  ...
  classCategory   nil
  subclasses      nil
topaz 1>
```

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

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

1.7 Escaping to an Editor

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

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

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

```
topaz 1> edit last
```

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

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

See "Editing Methods" on page 31 for more on **edit**.

1.8 Controlling the Display of Results

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

Display Level

When Topaz displays a result object, it ordinarily prints the name and value of each of the object's instance variables.

```
topaz 1> printit
Animal
%
Animal class
  superClass      Object class
  format          0
  instVars        3
  instVarNames    an Array
  constraints      an Array
  classVars       a SymbolDictionary
  methodDict      a GsMethodDictionary
  poolDictionaries an Array
  categories      a GsMethodDictionary
  secondarySuperclasses nil
  name            Animal
  ...
  classCategory   nil
  subclasses      nil
```

In other words, the default Topaz result display is one level deep. You can use the **level** command to ask for more or less information about results. Setting the level to 0 would give this view of Animal:

```
topaz 1> level 0
topaz 1> printit
Animal
%
Animal
```

The following example shows part of a two-level display:

```
topaz 1> level 2
topaz 1> printit
Animal
%
Animal class
  superClass      Object class
  superClass      nil
  format          0
  instVars        0
  instVarNames    an Array
  constraints      an Array
  classVars       a SymbolDictionary
  methodDict      a GsMethodDictionary
  poolDictionaries an Array
  categories      a GsMethodDictionary
  secondarySuperclasses nil
  name            Object
  classHistory    a ClassHistory
  description     a GsClassDocumentation
  migrationDestination nil
  timeStamp       a DateTime
  userId          SystemUser
  extraDict       a SymbolDictionary
  classCategory   nil
  subclasses      nil
format          0
instVars        3
instVarNames    an Array
  #1 name
  #2 favoriteFood
  #3 habitat
...
```

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

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

Setting Limits on Object Displays

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

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

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

Setting the limit to 0 restores the default condition.

Displaying Variable Names, OOPs, and Hex Byte Values

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

Instance Variable Names

As you saw in the display of class `Animal`, Topaz normally prints the name of each named instance variable with its value. If you don't need this information, you can

speed up the display of results by telling Topaz to **omit names**, as in the following example:

```
topaz 1> omit names
topaz 1> printit
Animal
%
Animal class
  i1 Object class
  i2 0
  i3 3
  i4 an Array
  i5 an Array
  i6 a SymbolDictionary
  i7 a GsMethodDictionary
  i8 an Array
  i9 a GsMethodDictionary
  i10 nil
  i11 Animal
  ...
```

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

Hexadecimal Byte Values

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

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

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

OOP Values

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

```
[151141 sz:6 cls: 867 Symbol]
```

Each object header contains:

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

For example:

```
topaz 1> display oops
topaz 1> printit
Animal
%
[150621 sz:19 cls: 150617 Animal class] Animal class
  superClass      [565 sz:19 cls: 1611 Object class] Object class
  format          [2 sz:0 cls: 581 SmallInteger] 0
  instVars        [14 sz:0 cls: 581 SmallInteger] 3
  instVarNames    [150613 sz:3 cls: 523 Array] an Array
  constraints      [150593 sz:3 cls: 523 Array] an Array
  classVars       [150589 sz:14 cls: 871 SymbolDictionary] a SymbolDictionary
  methodDict      [150633 sz:112 cls: 775 GsMethodDictionary] a GsMethodDictionary
  poolDictionaries [150561 sz:0 cls: 523 Array] an Array
  categories      [150637 sz:28 cls: 775 GsMethodDictionary] a GsMethodDictionary
  secondarySuperclasses [20 sz:0 cls: 597 UndefinedObject] nil
  name            [147877 sz:6 cls: 867 Symbol] Animal
...

```

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

1.9 Creating and Changing Methods

The first step in creating or editing a method is to tell Topaz the name of the method's class. Do this with the **set class** command:

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

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

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

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

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

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

```
topaz 1> status
```

Current settings are:

(display of current settings and connection information appears here)

browsing information:

```
Class_____ Animal
Category_____ Updating
```

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

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

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

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

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

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

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

Editing Methods

You can debug and refine methods by using Topaz's **edit** function in much the same way you use that function to create and modify **printit** commands. For

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

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

Here is how you would edit an existing class method:

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

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

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

1.10 Listing Methods and Categories

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

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

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

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

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

1.11 Committing and Aborting Transactions

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

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

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

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

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

1.12 Capturing Your Topaz Session In a File

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

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

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

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

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

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

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

The following example stops output to the current file:

```
topaz 1> output pop
```

As the command names **push** and **pop** imply, Topaz can maintain a stack of up to 20 output files. If you add the keyword **only** to the **push** command lines, current

interactions are captured only in the file on top of the stack. This prevents the results from showing on your screen, however.

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

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

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

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

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

1.13 Filing Out Classes and Methods

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

- transport your code to other GemStone systems,
- perform global edits and recompilations,

- produce paper copies of your work, and
- recover code that would otherwise be lost when you are unable to commit.

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

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

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

```
printit
Object subclass: 'Animal'
  instVarNames: #( 'name' 'favoriteFood' 'habitat' )
  classVars: #( )
  classInstVars: #( )
  poolDictionaries: #[]
  inDictionary: UserGlobals
  constraints: #[]
  instancesInvariant: false
  isModifiable: false
%
category: 'Updating'
method: Animal
habitat: newValue

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

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

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

fileout category:

Files out all the methods in the named category.

fileout classcategory:

Files out all the class methods in the named category.

fileout classmethod:

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

fileout method:

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

```
topaz 1> fileout habitat
```

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

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

1.14 Creating a Topaz Script for Batch Processing

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

Another way to produce such a script is to capture a typical Topaz session in a file, using **output push**. Edit the output file to remove the prompts and results, leaving only the Topaz commands and GemStone Smalltalk code. For example, suppose you wanted to make a script of the `mathtest2.log` file you created earlier. This is how it looks:

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

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

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

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

1.15 Taking Topaz Input from a File

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

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

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

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

1.16 Interrupting Topaz and GemStone

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

- When Topaz is awaiting input from your terminal, such as when you're entering a command, you can enter **Control-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 **Control-C** sends a request to GemStone to interrupt its activities as soon as possible. GemStone stops execution at the conclusion of the current method, and Topaz displays the message: `A soft break was received.`
- Typing **Control-C** three times immediately halts Topaz. Do this only in an emergency. All GemStone work performed since you last committed is lost.

1.17 Multiple Concurrent GemStone Sessions

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

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

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

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

```
topaz 1> printit
System currentSessionNames
%

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

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

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

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

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

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

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

1.18 Structural Access To Objects

In your GemStone Smalltalk programs, you should generally access the values stored in objects only by sending messages. During debugging, however, it's sometimes useful to be able to read an instance variable or store a value in it

without sending a message. For example, if an instance variable is normally read only by a message with side effects, it won't do to examine its value during debugging by sending that message.

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

Examining Instance Variables with Structural Access

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

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

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

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

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

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

As far as **at:** is concerned, named, indexed, and unordered instance variables are all numbered, with named instance variables appearing first, followed by indexed instance variables, then unordered instance variables. That is, if an indexed object also had three named instance variables, the first indexable field would be

addressed with **object at: 4**. Offsets into the unordered portions of NSCs are not consistent across `add:` or `remove:` commands.

Specifying Objects

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

Object Identity Specification Formats

@integer

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

integer

A 61-bit literal SmallInteger.

\$character

A literal Character.

aVariableName

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

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

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

Literal Object Specification Formats

'text'

A literal String.

#text

A literal Symbol (no white space allowed).

float

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

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

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

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

...
topaz 1> ! Look at first element of instVarNames array
topaz 1> object @1335297 at: 1
[1248257 sz:4 cls: 110849 Symbol] name
topaz 1> ! Look at first character of first instvarname
topaz 1> omit oops
topaz 1> object ** at: 1
$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, not `@1335297`.

1.19 Defining Local Variables

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

Creating Variables

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

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

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

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

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

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

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

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

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

Displaying Current Variable Definitions

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

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

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

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

Clearing Variable Definitions

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

For example:

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

1.20 Sending Messages

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

Therefore, Topaz provides the **send** command, which lets you send a message to an object identified by any of the means described in "Specifying Objects" on page 41. For example:

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

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

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

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

1.21 Logging Out

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

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

As noted above, logging out implicitly aborts your transaction.

1.22 Leaving Topaz

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

```
topaz> exit
```

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

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

Debugging Your GemStone Smalltalk Code

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

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

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

2.1 Step Points and Breakpoints

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

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

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

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

    <primitive: 94>

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

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

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

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

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

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

2.2 Setting, Clearing, and Examining Breakpoints

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

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

For example:

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

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

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

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

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

```

topaz 1> list breaks method: <
< aCharCollection

>Returns true if the receiver collates before the
argument. Returns false otherwise.

The comparison is case-insensitive unless the receiver
and argument are equal ignoring case, in which case
upper case letters collate before lower case letters.
The default behavior for SortedCollections and for
the sortAscending method in UnorderedCollection is
consistent with this method, and collates as follows:

#( 'c' 'MM' 'Mm' 'mb' 'mM' 'mm' 'x' ) asSortedCollection

yields the following sort order:

'c' 'mb' 'MM' 'Mm' 'mM' 'mm' 'x'
"

<primitive: 28>

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

```

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

```

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

```

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

```

topaz 1> break disable 2

```

A similar command line reenables the break point:

```

topaz 1> break enable 2

```

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

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

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

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

2.3 Examining the GemStone Smalltalk Call Stack

You can display all of the contexts in the active call stack by issuing the **stack** command with no arguments. Using the **stack** command displays method arguments and temporaries; the display is controlled by the current **level** setting. Use the **stk** command to display a summary call stack, with one line for each context.

Here's an example of the stack display for **level 0**, when **display oops** is active:

```
topaz 1> display oops
topaz 1> stack
1 Behavior >> new @ 1    [GsMethod 10941]
  receiver [208201 sz:19 cls: 208173 Animal class] Animal class
2 Executed Code @ 2    [GsMethod 208709]
  receiver [10 sz:0 cls: 1193 UndefinedObject] nil
  aDog [10 sz:0 cls: 1193 UndefinedObject] nil
```

Here's the equivalent display when **omit oops** is active:

```
topaz 1> omit oops
topaz 1> stack
1 Behavior >> new @ 1
  receiver Animal class
2 Executed Code @ 2
  receiver nil
  aDog nil
```

With **display oops** active, this is the display for the **stk** command:

```
topaz 1> stk
1 Behavior >> new @ 1    [GsMethod 10941]
2 Executed Code @ 2    [GsMethod 208709]
```

The display of each context includes:

- the level number of the context (for subsequent use with the **stack scope** command, described later);
- the OOP of the GsMethod (if **display oops** is active);
- the class of the receiver (and its OOP, if **display oops** is active);
- the class of the method invoked;
- the selector of the method;
- the current step point within the method, if any (an integer); and
- for the stack command, parameters and temporaries for this context (including 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**.

In the example stack list given above, an invocation of the method `new` is found at the top of the stack. The message was sent to the class `Animal`. Execution of the method was halted at step point 1.

The next item on the stack is code that was compiled by the `printit` command; this has an undefined receiver. Execution of this method halted during the call to the method `new`, which was at step point 2.

Proceeding After a Breakpoint

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

continue

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

step over

Tells GemStone Smalltalk to advance execution to the next step point (message-send, assignment, etc.) in the active context or its caller, and halt. The active context is the context specified by the last **stack scope**, **stack up**, or **stack down** command, or otherwise the top of the stack.

step into

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

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

Examining and Modifying Temporaries and Arguments

The Topaz **temporary** command lets you examine or modify the values of temporaries in the active context. If, for example, the method under inspection had a temporary variable named `count`, you could obtain its value by typing **temporary** and the variable name:

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

which returns a count of 5 in this example. Similarly, you can use the **temporary** command to assign a new value to a temporary variable:

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

When program execution pauses at a breakpoint, Topaz adds some temporaries to your local scope. The nature of these temporaries depends on the type of the current expression:

return

Topaz creates a temporary called `_returnValue` that shows the value to be returned.

assignment

Topaz creates `_newValue`, which holds the value to be assigned.

message-send

Topaz creates `_receiver` to show the receiver, and argument temporaries called `_arg1`, `_arg2`, and so on, to hold argument values.

The **temporary** command displays the values of these Topaz-created temporaries, when they exist.

Select a Context for Examination and Debugging

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

```
topaz 1> stack
1 Behavior >> new @ 1
  receiver Animal class
2 Executed Code @ 2
  receiver nil
  aDog nil
```

To show the active context, type:

```
topaz 1> stack scope
1 Behavior >> new @ 1
```

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

```
topaz 1> stack scope 2
2 Executed Code @ 2
```

Now confirm that Topaz redefined the active context:

```
topaz 1> stack scope
2 Executed Code @ 2
```

Redefine the Active Call Stack

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

```
topaz 1> stack all
*1 Behavior >> new @ 1
 2 Animal >> habitat @ 3
 3 Executed Code @ 2
```

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 2  
Stack 2 selected  
1 Animal >> habitat @ 3  
2 Executed Code @ 3
```

—
|

Command Dictionary

This chapter provides brief descriptions of the Topaz commands for quick reference. The commands are presented in alphabetical order.

Command Syntax

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

Topaz commands are case-insensitive. Thus, **Time**, **TIME**, and **time** are regarded 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, because GemStone Smalltalk, the language of category names, is case-sensitive. The same is true of UNIX path names, in commands such as **output push myFile.out**, and UNIX user names and passwords.

In general, objects passed as arguments to Topaz commands can be specified using any of the formats described in “Specifying Objects” on page 41.

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

—
|

ABORT

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

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

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

BEGIN

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

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

```
System transactionMode: #manualBegin
```

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

BREAK *aSubCommand*

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

Method Breakpoints

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

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

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

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

@integer

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

aVariableName

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

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

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

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

Establishes a method breakpoint on the given instance method.

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

Establishes a method breakpoint on the given class method.

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

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

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

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

Displaying Breakpoints

break list

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

Disabling and Enabling Breakpoints

break disable *anIndex*

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

break disable all

Disables all currently set breakpoints.

break enable *anIndex*

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

break enable all

Reenables all disabled breakpoints.

Deleting Breakpoints

break delete *anIndex*

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

break delete all

Deletes all currently set breakpoints.

Examples

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

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

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

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

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

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

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

```
topaz 1> break list
1: GsFile >> nextLine @ 1
2: GsFile class >> openRead: @ 2
3: String >> < @ 2
topaz 1> break disable 2
topaz 1> break list
1: GsFile >> nextLine @ 1
2: GsFile class >> openRead: @ 2 (disabled)
3: String >> < @ 2
topaz 1> break enable 2
topaz 1> break list
1: GsFile >> nextLine @ 1
2: GsFile class >> openRead: @ 2
3: String >> < @ 2
topaz 1> break delete 1
topaz 1> break list
2: GsFile class >> openRead: @ 2
3: String >> < @ 2
topaz 1> break delete all
topaz 1> break list
No breaks set
```

CATEGORY: *aCategoryName*

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

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

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

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

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


CLASSMETHOD[: *aClassName*]

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

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

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

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

COMMIT

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

—
|

CONTINUE [*anObjectSpec*]

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

continue

Attempts to continue execution.

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 41.

For more information about breakpoints, see the discussion of the **break** command on page 61, or see Chapter 2, “Debugging Your GemStone Smalltalk Code”.

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

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

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

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

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

All Topaz object specification formats (as described in “Specifying Objects” on page 41) are legal in **define** commands.

define

Lists all current local variable definitions.

define *aVarName*

Deletes the definition of the variable *aVarName*.

define *aVarName anObjectSpec aSelectorOrArg ...*

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

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

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

DISASSEM [*aClassParameter*] *aParamValue*

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

If the session is remote, the output goes to stdout of the remote Gem, which is the gem log.

disassem *@anOop*

Disassemble the method or code object with the specified oop.

disassem method: *aSelector*

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

disassem classmethod: *aSelector*

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

DISPLAY *aDisplayFeature*

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

display oops

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

display bytes

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

display errorcheck

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

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

display names

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

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

display resultCheck

Allows Topaz programs to check input values and record the results. This command creates the `./topazerrors.log` file or opens the file to append to it, if it already exists. Specifying **display resultCheck** is equivalent to setting **expectvalue true**, except that it affects the behavior of all **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 **printit**

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

display pauseonerror

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

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

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

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

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

```
display interactive pause on errors
```

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

DOIT

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

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

If you use this command to execute a GemStone Smalltalk host file access method, such as `GsFile class>>openRead:` or `openWrite:`, and you do not supply an explicit path specification as part of the method argument, the default directory for the method depends on the version of the Topaz that you are running. With linked Topaz, the default directory is the directory in which Topaz was started. With RPC Topaz, the default directory is the `$HOME` directory of the hostuser account.

DOWN [*anInteger*]

Moves the scope down within the currently selected call stack, and displays the activation selected as a result. The optional argument *anInteger* specifies how many activations to move down. If no argument is supplied, the scope will go down one activation. See also **stack down** on page 130.

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

```
topaz 1> stk
1 System class >> signal:args:signalDictionary: @8 line 15
2 Object >> _error:args: @7 line 10
3 Object >> _error: @2 line 5
4 Number >> _errorDivideByZero @1 line 5
==> 5 SmallInteger >> / @4 line 6
6 Executed Code @2 line 1
topaz 1> down
4 Number >> _errorDivideByZero @1 line 5
  receiver 5
```

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 **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 **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 **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 **printit** command with the new text.

edit new text

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

Creating or Modifying GemStone Smalltalk Methods

edit new

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

edit new method

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

edit new classmethod

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

edit aSelector**edit method:** *aSelector*

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

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

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

edit classmethod: *aSelector*

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

ERRORCOUNT

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

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

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

```
topaz 1> object ErrorCount
```

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

EXIT [*anInteger*]

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

An optional integer argument allows you to indicate an error status. A zero (0) indicates successful status; other integers indicate an error and will result in a return code of three (3).

EXPECTBUG *bugNumber*

value *resultSpec* [*integer*] |
error *errCategory* *errNumber* [*resultSpec* [*resultSpec*]..]

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

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

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

resultSpec is specified as in the **expectvalue** command (page 82).

errCategory and *errNumber*
 are specified as in the **expecterror** command (page 79).

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

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

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

EXPECTERROR *anErrorCategory anErrorNumber*

[*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 **printit**, **doit**, **method:**, **classmethod:**, **commit**, and **send**.

anErrorCategory must be the object identifier of an error category, and *anErrorNumber* must be a SmallInteger. For example:

```
GemStoneError 2010
```

All Topaz object specification formats (as described in “Specifying Objects” on page 41) are legal in **expecterror** commands. In addition, this command takes two more formats that allow you to specify instances of classes as error arguments:

%className An instance of the class *className*.

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

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

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

If you care about the number of error arguments, put that many *anErrorArg* tokens on the command line. The error count is incremented if the actual error contains fewer arguments than were specified in the **expecterror** command; it is not incremented if the error contains more arguments than specified. If you don't care about the class of the error arguments, specify */Object* for each one.

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

```
topaz 1> errorcount
0
topaz 1> expecterror GemStoneError 2010 1 /Symbol
topaz 1> printit
1 x
%
-----
GemStone: Error          Nonfatal
No method was found for the selector #x when sent to 1
with arguments
contained in anArray( ).
Error Category: [GemStone] Number: 2010 Arg Count: 3
Arg 1: 1
Arg 2: x
Arg 3: an Array
topaz 1> errorcount
0
topaz 1>
```

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


```
topaz 1> errorcount
0
topaz 1> iferror stack
topaz 1> expecterror GemStoneError 2010 1 /Symbol
topaz 1> printit
'abc' at: 5
%
-----
GemStone: Error           Nonfatal
An indexable object or NSC 'abc' was referenced with an
index 5 that was out of range.
Error Category: [GemStone] Number: 2003 Arg Count: 2
Arg 1: abc
Arg 2: 5
ERROR: WRONG ERROR CATEGORY/NUMBER
Now executing the following command saved from "iferror":
  stack
1 System class >> signal:args:signalDictionary: @9 line15
  receiver System
  anInteger 2003
  anArray an Array
    #1 abc
    #2 5
  anErrorDict a SymbolDictionary
...
topaz 1> errorcount
1
topaz 1>
```

EXPECTVALUE *anObjectSpec* [*anInt*]

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

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

All Topaz object specification formats (as described in “Specifying Objects” on page 41) are legal in **expectvalue** commands. In addition, this command takes further formats that allow you to specify instances of classes:

%className

An instance of the class *className*.

%@OOPOfClass

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

/className

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

/@OOPOfClass

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

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

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

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

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

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

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

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

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

FILEOUT *aSubCommandOrSelector* [TOFILE: *aFileName*]

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

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

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

fileout class: [*aClassName*]

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

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

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

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

fileout category: *aCategoryName*

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

fileout classcategory: *aCategoryName*

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

fileout classmethod: *aSelector*

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: *aSelector*

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 *aSelector*

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

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

FRAME [*anInteger*]

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

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

See also **stack scope** on page 130, the `up` command on page 140 and the `down` command on page 73.

For example,

```
topaz 1> stk
==> 1 System class >> signal:args:signalDictionary: @8 line 15
2 Object >> _error:args: @7 line 10
3 Object >> _error: @2 line 5
4 Number >> _errorDivideByZero @1 line 5
5 SmallInteger >> / @4 line 6
6 Executed Code @2 line 1
topaz 1> frame 5
5 SmallInteger >> / @4 line 6
  receiver 4
  aNumber 0
topaz 1> stk
1 System class >> signal:args:signalDictionary: @8 line 15
2 Object >> _error:args: @7 line 10
3 Object >> _error: @2 line 5
4 Number >> _errorDivideByZero @1 line 5
==> 5 SmallInteger >> / @4 line 6
6 Executed Code @2 line 1
topaz 1>
```

GCITRACE *aFileName*

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

—
|

HELP [*aTopicName*]

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

To display help text for **fileout**:

```
topaz 1> help fileout
```

To display help text for **last**:

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

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

IFERR *bufferNumber* [*aTopazCommandLine*]

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

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

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

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

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

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

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

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

The **iferr_list** command will display the contents of all post-error buffers.

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

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


IFERR_CLEAR

The `iferr_clear` command clears all the post-error command buffers.

For details on the post-error command buffers, see the `iferr` command on page 88.

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 88.

—
|

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 88.

INPUT [*aFileName* | POP]

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

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

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

input *aFileName*

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

input pop

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

LEVEL *anIntegerLevel*

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

LIMIT [BYTES | OOPS] *anInteger*

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

limit *anInteger*

limit bytes *anInteger*

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

limit oops *anInteger*

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

LIST

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

Browsing Dictionaries and Classes

list dictionaries

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

list classesIn: *aDictionary*

Lists the classes in *aDictionary*. For example,

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

lists all of the classes in your UserGlobals dictionary.

list classes

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

list categoriesin: [*aClass*]

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

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

Listing Methods

list aSelector

list method: *aSelector*

Lists the source code of the specified instance method for the current class.

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

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

list classmethod: *aSelector*

Lists the category and the source code of the specified class method for the current class.

list

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

Listing Step Points**list steps**

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

list steps method: aSelector

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

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

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

  <primitive: 94>

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

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

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

list steps classmethod: aSelector

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

Listing Breakpoints

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

list breaks

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

list breaks method: *aSelector*

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

```
topaz 1> list breaks method: <
< aCharCollection
```

```
"Returns true if the receiver collates before the
argument. Returns false otherwise.
```

```
The comparison is case-insensitive unless the receiver
and argument are equal ignoring case, in which case
upper case letters collate before lower case letters.
The default behavior for SortedCollections and for
the sortAscending method in UnorderedCollection is
consistent with this method, and collates as follows:
```

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

```
 yields the following sort order:
```

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

```
"
```

```
<primitive: 28>
```

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

list breaks classmethod: *aSelector*

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

LOADUA *aFileName*

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

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

LOGIN

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

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

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

set username:
your GemStone user ID.

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

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

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

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

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

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

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

For more information about logging in to GemStone, see the description of **set** on page 121. Also see the section of Chapter 1 entitled "Logging In to GemStone."

—
|

LOGOUT

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

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

LOOKUP

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

Finding and Listing Methods

lookup classmethod *aSelector*

Lists the source code of the specified class method for the current class, or searching the superclasses, the first superclass that implements this method.

lookup method *aSelector*

Lists the source code of the specified instance method for the current class, or searching the superclasses, the first superclass that implements this method.

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

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

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

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

METHOD[: *aClassName*]

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

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

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

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

NBRUN

Similar to **printit**, 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, see **nbresult**.

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

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.

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

NBRESULT

Wait for and display the result of a previous **nbrun** call. This call must be preceded by a **set session** to switch to the session of an outstanding **nbrun**. 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:

```
[Oop: 0 Object Does Not Exist]
```

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

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 41 are legal in **object** commands.

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

object *anObjectSpec* **at:** *anIndex*

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

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

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

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

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

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

CAUTION

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

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

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

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

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

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

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

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

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

Character. This stores the character '9':

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

SmallInteger. This stores a byte with the value 48:

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

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

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

OMIT *aDisplayFeature*

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

omit oops

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

omit bytes

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

omit errorcheck

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

omit names

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

omit resultCheck

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

omit pauseonerror

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

```
omit interactive pause on errors
```

OPAL

Included for compatibility with previous versions. See the **doit** command on page 72.

OUTPUT (PUSH | APPEND | PUSHNEW | POP) *aFileName* [ONLY]

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

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

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

output *aFileName*

output push *aFileName*

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

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

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

output append *aFileName*

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

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

output pushnew *aFileName*

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

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

output *aFileName* **only**

output push *aFileName* **only**

output append *aFileName* **only**

output pushnew *aFileName* **only**

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

output pop

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

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

PRINTIT

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

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

Executing GemStone Smalltalk Host File Access Methods

If you use this command to execute a GemStone Smalltalk host file access method, and you do not supply an explicit path specification as part of the method argument, the default directory for the GemStone Smalltalk method depends on the version of the Topaz that you are running. With linked Topaz, the default directory is the directory in which Topaz was started. With the RPC version, the default directory is the \$HOME directory of the hostuser account.

PROTECTMETHODS

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

Used for consistency checking in filein scripts.

QUIT [*anInteger*]

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

An optional integer argument allows you to indicate an error status. A zero (0) indicates successful status; other integers indicate an error and will result in a return code of three (3).

RELEASEALL

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

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

REMARK *commentText*

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

REMOVEALLMETHODS[: *aClassName*]

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

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

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

REMOVEALLCLASSMETHODS[: *aClassName*]

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

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

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

RUN

Included for compatibility with previous versions. See the **printit** command on page 112.

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 41) are legal in **send** commands.

SET *aTopazParameter* [*aParamValue*]

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

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

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

set cachename: *aString*

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

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

set category: *aCategory*

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

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

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

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

set class: *aClassName*

Sets the current class. You must be logged in to use this command. After setting the current class, you can list its categories and methods with the **list categories** command. You can select a category to work with through either the **set category:** or **category:** command.

The current class may also be redefined by the **list categoriesin:**, **method:**, **classmethod:**, **removeallmethods:**, **removeallclassmethods:**, and **fileout class:** commands.

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

set editorname: *aHostName*

Sets the name of the editor you want to use in conjunction with the **edit** command. For example:

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

The default is set from your \$EDITOR environment variable, if it is defined.

set gemnetid: *aServiceName*

aServiceName is a network resource string specifying the name of the GemStone service (that is, the host process to which your Topaz session will be connected) and its host computer.

For the RPC version of Topaz the default gemnetid parameter is `gemnetobject`, which is the GemStone service name in most GemStone installations. However, if you use the UNIX C shell (`/bin/csh`) on the given Gem network node, specify the GemStone service name `gemnetobjcsh` instead of `gemnetobject`.

For a linked Topaz session, the default is `gcilnkobj`. Before you log in, use the `status` command to make sure that this parameter is `gcilnkobj` or "" (null). This causes **topaz -l** to make the first session a linked session. If **gemnetid** is set to anything else, **topaz -l** starts RPC sessions. In this case, the prompt for the first session is `topaz 2>`, because `topaz 1>` is reserved for a linked session. After you start the RPC session you can still start a linked session by resetting the **gemnetid** to nil:

```
set gemnetid: ''
```

or to `gcilnkobj`. Once you have a linked session, any additional sessions are RPC, regardless of the **gemnetid** setting.

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:

```
!tcp@lichen!gemnetobject
```

To specify a Gem running on the current node, omit the `protocol@node` portion of the string, and specify only the Gem name: `gemnetobject`. Appendix B describes network resource string syntax.

set gemstone: *aGemStoneName*

Specifies the name of the GemStone you want to log in to. The standard name is `gs64stone`; if this doesn't work for you, see your GemStone data curator.

You can run your GemStone session (Gem), repository monitor (Stone) process, and your Topaz processes on separate nodes in your network. The one exception is the linked Topaz session, when Topaz and the Gem run as a single process. Network resource strings allow you to designate the nodes on which the Gem and Stone processes run. For example, a Stone process called `gs64stone` on node `lichen` could be described in network resource string syntax as:

```
!tcp@lichen!gs64stone
```

To specify a Stone running on the same node as the Gem, omit the *protocol@node* portion of the string, and specify only the Stone name: `gs64stone`. Appendix B describes network resource string syntax.

set hostpassword: *aPassword*

Sets the host password to be used when you next log in. If you don't include the password on the command line, Topaz prompts you for it. Prompted input taken from the terminal is not echoed. This lets you put a **set hostpassword:** command in your Topaz initialization file so that Topaz automatically prompts you for your password. Note, however, that this command must *follow* the **set hostusername:** command.

With TCP/IP, if you do not explicitly supply a host username and password, Topaz will try to find a username for the designated node in a file in your home directory. See the following discussion of **set hostusername**.

For a linked Topaz session, **set hostpassword** has no effect, because no separate Gem process is created on the host computer. The password is required, however, if you spawn new sessions while you are running linked Topaz, because the additional sessions are always RPC Topaz.

set hostusername: *aUsername*

Sets the account name you use when you log in to the host computer. When you run Topaz, a Gem (GemStone session) process is started on the host computer specified by the **set gemnetid:** command. The **set hostusername:** command tells Topaz which account you want that process to run under.

With TCP/IP, if you do not explicitly supply a host username and password, Topaz tries to find a username and password for the designated node in a

network initialization file in your home directory. Under UNIX, that file is `$HOME/.netrc` and should contain lines of the form

```
machine aNode login aUsername password aPassword
```

For example, `$HOME/.netrc` under UNIX:

```
machine alf login joejob password mypassword
```

Because the network initialization file contains your password, you should ensure that others — group or world — do not have authorization to read it.

To clear the `hostusername` field, enter:

```
topaz 1> set hostusername *
```

For a linked Topaz session, **set hostusername** has no effect. (No separate Gem process is created on the host computer.) It is required, however, if you spawn new sessions while you are running linked Topaz, because the additional sessions are always RPC Topaz.

set nrsdefaults: *aNRShheader*

Sets the default components to be used in network resource string specifications. The parameter *aNRShheader* is a network resource string header that may specify any NRS modifiers' default values. The initial value of `nrsdefaults` is the value of the `GEMSTONE_NRS_ALL` environment variable. The Topaz **status** command shows the value of **nrsdefaults** unless it is the empty string.

set password: *aGemStonePassword*

Sets the GemStone password to be used when you next log in. If you don't include the password on the command line, Topaz prompts you for it. Prompted input is taken from the terminal and not echoed. This lets you put a **set password:** command in your Topaz initialization file so that Topaz will automatically prompt you for your password. Note, however, that this command must *follow* the **set username:** command.

set session: *aSessionNumber*

Connects Topaz to the session whose ID is *aSessionNumber*. When you log in to GemStone, Topaz displays the session ID number for that connection. This command allows you to switch among multiple sessions. (The Topaz prompt always shows the number of the current session.)

If you specify an invalid session number, an error message is displayed, and the current session is retained.

This command clears the current class and category. After you switch sessions with **set session**, your local variables (created with the **define** command) no longer have valid definitions.

set sourcestringclass: *aClass*

Sets the class used to instantiate Smalltalk source strings generated by the **run**, **edit**, **method**, and **classmethod** commands. For example:

```
set sourcestringclass String
set sourcestringclass DoubleByteString
set sourcestringclass IsoLatin
```

The Topaz **status** command shows the current source class. The default source class is String. The **set session** command resets the source string class to the default.

set username: *aGemStoneUsername*

Establishes a GemStone user ID for the next login attempt. Your GemStone data curator can tell you your user name.

SHELL [*aHostCommand*]

When issued with no parameters, this command creates a child process in the host operating system, leaving you at the operating system prompt. To get back into Topaz, exit the command shell by typing **Control-D** (from the UNIX Bourne or Korn shells), typing **logout** (from the UNIX C shell), or typing **exit** (from a DOS shell).

If you supply parameters on the **shell** command line, they pass to a subprocess as a command for execution. For example:

```
topaz 1> shell ls -l /user1/janec.topaz
total 4
-rw-r--r-- 1 janec 196 Jul 1 22:31 animal.gs
-rw-r--r-- 1 janec 139 Jul 1 22:31 animaltest.log
-rw-r--r-- 1 janec 287 Jul 1 22:31 mathtest.log
-rw-r--r-- 1 janec 110 Jul 1 22:32 mathtest2.log
topaz 1>
```

On UNIX systems, a **shell** command issued without parameters creates a shell of whatever type is customary for the user account (C, Bourne, or Korn). When issued with parameters, **shell** always creates a shell of the system default type (either Bourne or Korn).

SPAWN [*aHostCommand*]

Included for compatibility with previous versions. See the **shell** command on page 126.

STACK [*aSubCommand*]

Topaz can maintain up to eight simultaneous GemStone Smalltalk call stacks that provide information about the GemStone state of execution. Each call stack consists of a linked list of contexts.

The call stack becomes active, and the **stack** command becomes accessible, when you execute GemStone Smalltalk code containing a breakpoint. The **stack** command allows you to display the contexts in the active call stack; redefine the active context for debugging commands within the active call stack; reset whether the active call stack is automatically saved before certain Topaz execution commands are performed; display the top context of each saved call stacks; redefine the active call stack; or remove one or all call stacks.

Display the Active Call Stack

stack

Displays all of the contexts in the active call stack, starting with the active context. For each context in the stack display, the following items are displayed:

- the OOP of the GsMethod (if **display oops** is active)
- the class of the receiver (and its OOP, if you have specified **display oops**)
- the class of the GsMethod
- selector of the method
- its level number (as used in the **stack scope** command)
- the current step point (that is, assignment, message send, or method return) within the method (an integer, as in **list steps**)
- parameters and temporaries for this context (including OOPs, if you have specified **display oops**)

If any context in the display is a block, the display for that context begins with SimpleBlock or ComplexBlock, as shown in the examples below.

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

The active context resets to 1 whenever any of the following commands is executed: **printit**, **send**, **doit**, **step**, **edit last**, or **edit new text**.

Here is an example of the stack display when **display oops** is active:


```
topaz 1> run
#[ 1, 2] do[:x | x / 0 ]
%
-----
GemStone: Error          Nonfatal
An attempt was made to divide 1 by zero.
Error Category: 231169 [GemStone] Number: 2026 Arg Count: 1 Context :
1968897
Arg 1: [10 sz:0 cls: 74241 SmallInteger] 1
topaz 1> lev 0
topaz 1> display oops
topaz 1> stack
1 System class >> signal:args:signalDictionary: @8 line 15
[GsMethod 35394817]
  receiver [76033 sz:19 cls: 1048065 System class] System
  anInteger [16210 sz:0 cls: 74241 SmallInteger] 2026
  anArray [1966593 sz:1 cls: 66817 Array] anArray
  anErrorDict [231169 sz:50 cls: 111361 SymbolDictionary]
aSymbolDictionary
  ex [20 sz:0 cls: 76289 UndefinedObject] nil
  _topOfStack [76033 sz:19 cls: 1048065 System class] System
2 Object >> _error:args: @7 line 10 [GsMethod 13990657]
  receiver [10 sz:0 cls: 74241 SmallInteger] 1
  errorSymbol [4250881 sz:23 cls: 110849 Symbol]
numErrIntDivisionByZero
  argList [233217 sz:0 cls: 66817 Array] anArray
  newArgList [1966593 sz:1 cls: 66817 Array] anArray
  errorNumber [16210 sz:0 cls: 74241 SmallInteger] 2026
3 Object >> _error: @2 line 5 [GsMethod 35717121]
  receiver [10 sz:0 cls: 74241 SmallInteger] 1
  errorSym [4250881 sz:23 cls: 110849 Symbol]
numErrIntDivisionByZero
4 Number >> _errorDivideByZero @1 line 5 [GsMethod 14009089]
  receiver [10 sz:0 cls: 74241 SmallInteger] 1
5 SmallInteger >> / @4 line 6 [GsMethod 35609345]
  receiver [10 sz:0 cls: 74241 SmallInteger] 1
  aNumber [2 sz:0 cls: 74241 SmallInteger] 0
6 SimpleBlock in Executed Code @3 line 1 [GsMethod 1954561]
  self [20 sz:0 cls: 76289 UndefinedObject] nil
  receiver [1953793 sz:9 cls: 84993 SimpleBlock] aSimpleBlock
  x [10 sz:0 cls: 74241 SmallInteger] 1
7 Collection >> do: @5 line 10 [GsMethod 20214273]
```

```

receiver [1949953 sz:2 cls: 66817 Array] anArray
aBlock [1953793 sz:9 cls: 84993 SimpleBlock] aSimpleBlock
i [10 sz:0 cls: 74241 SmallInteger] 1
_temp1 [10 sz:0 cls: 74241 SmallInteger] 1
_temp2 [18 sz:0 cls: 74241 SmallInteger] 2
_temp3 [10 sz:0 cls: 74241 SmallInteger] 1
8 Executed Code @4 line 1 [GsMethod 1954561]
receiver [20 sz:0 cls: 76289 UndefinedObject] nil

```

stack anInt

Displays contexts in the active call stack, starting with the active context. The argument *anInt* indicates how much of the stack to display. For example, if *anInt* is 1, this command shows only the active context. If *anInt* is 2, this command also shows the caller of the active context, etc.

Display or Redefine the Active Context

stack scope

Displays the current scope, starting with the active context. For example:

```

topaz 1> stack scope
1 Behavior >> new @ 1

```

stack scope anInt

Redefines the active context within the active call stack and displays the new context. The integer 1 represents the currently active context, while the integer 2 represents the *caller* of the active context.

stack up

Moves the current scope up one level toward the top of the stack and displays the new context. This is equivalent to the command line:

```
stack scope <current scope + 1>
```

stack down

Moves the current scope down one level away from the top of the stack and displays the new context. This is equivalent to the command line:

```
stack scope <current scope - 1>
```

stack trim

Trims the stack so that the current scope becomes the new top of the stack. Execution resumes at the first instruction in the method at the new top of the stack. If that method has been recompiled, **stack trim** installs the new version

of the method. The new top of the stack must not represent the activation of an ExecutableBlock.

Save or Delete the Active Call Stack During Execution

stack nosave

Causes your active call stack to be deleted before executing any of the following commands: **printit**, **send**, **doit**, **edit last**, or **edit new text**.

This is the default condition.

stack save

Automatically saves the active call stack before executing any of the commands listed for **stack nosave** (above).

Display All Call Stacks

stack all

Displays your list of saved call stacks. The list includes the top context of every call stack (stack 1). For example:

```
topaz 1> stack all
*1 Behavior >> new @ 1
 2 Animal >> habitat @ 3
 3 Executed Code @ 2
```

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

Redefine the Active Call Stack

stack change *anInt*

Sets the active call stack to the call stack indicated by *anInt* in the **stack all** command output, and displays one frame of the newly selected call stack.

Remove Call Stacks

stack delete *aStackInt*

Removes the call stack indicated by *aStackInt* in the **stack all** command output.

Topaz maintains up to eight simultaneous call stacks. If all eight call stacks are in use, you must use this command to delete a call stack before issuing any of the following commands: **printit**, **send**, **doit**, **edit last**, or **edit new text**.

stack delete all

Removes all call stacks.

STATUS

Displays your current login settings and other information about your Topaz session.

For example:

```
topaz 1> status
Current settings are:
  display level: 1
  omit oops
  omit bytes
  display instance variable names
  omit automatic result checks
  omit interactive pause on errors
EditorName_____ gnumacs -nw
Connection Information:
UserName_____ 'Isaac Newton'
Password _____ (set)
HostUserName_____ 'newtoni'
HostPassword_____ (set)
GemStone_____ 'gs64stone'
GemNetId_____ 'gemnetobject'
GemStone
NRS_____ '!#auth:newtoni@password#server!gs64stone'
browsing information:
Class_____
Category_____ (as yet unclassified)
Source String Class__ String
```

STEP (OVER | INTO)

Advances execution to the next step point (assignment, message send, or method return) and halts. You can use the step command to continue execution of your GemStone Smalltalk code after an error or breakpoint has been encountered. For examples and other useful information, see Chapter 2, “Debugging Your GemStone Smalltalk Code.”

step

Equivalent to **step over**.

step over

Advances execution to the next step point in the active context or its caller. The active context is the top of the stack or the context specified by the last **stack scope**, **stack up**, or **stack down** command.

step into

Advances execution to the next step point in your GemStone Smalltalk code.

STK [*aSubCommand*]

Similar to **stack**, but does not display parameters and temporaries for each context. All contexts for the active call stack are displayed, with the current active context indicated by an arrow.

For more information on **s**, see the **stack** command on page 128.

```
topaz 1> run
#[ 1, 2] do:[:x | x / 0 ]
%
-----
GemStone: Error          Nonfatal
An attempt was made to divide 1 by zero.
Error Category: [GemStone] Number: 2026 Arg Count: 1
Arg 1: 1
topaz 1> stk
1 System class >> signal:args:signalDictionary: @8 line 15
2 Object >> _error:args: @7 line 10
3 Object >> _error: @2 line 5
4 Number >> _errorDivideByZero @1 line 5
5 SmallInteger >> / @4 line 6
6 SimpleBlock in Executed Code @3 line 1
7 Collection >> do: @5 line 10
8 Executed Code @4 line 1
topaz 1> up 4
5 SmallInteger >> / @4 line 6
  receiver 1
  aNumber 0
topaz 1> stk
1 System class >> signal:args:signalDictionary: @8 line 15
2 Object >> _error:args: @7 line 10
3 Object >> _error: @2 line 5
4 Number >> _errorDivideByZero @1 line 5
==> 5 SmallInteger >> / @4 line 6
6 SimpleBlock in Executed Code @3 line 1
7 Collection >> do: @5 line 10
8 Executed Code @4 line 1
```

TEMPORARY [*aTempName*[/*anInt*] [*anObjectSpec*]]

Displays or redefines the value of one or more temporaries in the active context previously specified by a **stack** or **stack scope** command. The **stack 1** command shows the currently active context. For examples and other useful information, see Chapter 2, “Debugging Your GemStone Smalltalk Code.”

All Topaz object specification formats (as described in “Specifying Objects” on page 41) are legal in **temporary** commands.

temporary

Displays the names and values of all temporary objects in the active context.

temporary *aTempName*

Displays the value of the first temporary object with the specified name in the active context.

```
topaz 1> temporary preferences
preferences      an Array
```

temporary *aTempName anObjectSpec*

Redefines the specified temporary in the active context to have the value *anObjectSpec*.

temporary *anInt*

Displays the value of the temporary at offset *n* in the active context. Use this form of the command to access a temporary with a duplicate name, because **temporary** *aTempName* always displays the first temporary with the specified name.

temporary *anInt anObjectSpec*

Redefines the temporary at offset *n* in the active context to have the value *anObjectSpec*.

Temporaries displayed as `_temp#` are un-named temporaries private to the virtual machine, like the temporaries used in evaluation of the optimized `to:do:`, shown in the example below.

```
topaz 1> run
| a |
1 to: 25 do:[:j | a := j . a pause ]
%
Execution has been suspended by a "pause" message.
Topaz pausing after error, type <return> to continue, ctl-C
to quit ?
topaz 1> stack
1 Object >> pause @2 line 8    [GsMethod 35739393]
  receiver [10 sz:0 cls: 74241 SmallInteger] 1
2 Executed Code @3 line 2    [GsMethod 1328129]
  receiver [20 sz:0 cls: 76289 UndefinedObject] nil
  a [10 sz:0 cls: 74241 SmallInteger] 1
  j [10 sz:0 cls: 74241 SmallInteger] 1
  _temp1 [10 sz:0 cls: 74241 SmallInteger] 1
  _temp2 [202 sz:0 cls: 74241 SmallInteger] 25
  _temp3 [10 sz:0 cls: 74241 SmallInteger] 1
```

TIME

Displays the current time from the system clock.

—
|

UNPROTECTMETHODS

Cancels the effect of **protectmethods**, which is used for consistency checking in filein scripts.

UP [*anInteger*]

Moves the scope up within the currently selected call stack, and displays the activation selected as a result. The optional argument *anInteger* specifies how many activations to move up. If no argument is supplied, the scope will go up one activation. See also **stack up** on page 130.

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

```
topaz 1> stk
1 System class >> signal:args:signalDictionary: @8 line 15
2 Object >> _error:args: @7 line 10
3 Object >> _error: @2 line 5
==> 4 Number >> _errorDivideByZero @1 line 5
5 SmallInteger >> / @4 line 6
6 Executed Code @2 line 1
topaz 1> up
5 SmallInteger >> / @4 line 6
   receiver 5
   aNumber 0
```

WHERE [*anInteger*]

Displays the current call stack, with one line per activation. If no argument is specified, all lines of the current call stack are displayed, equivalent to **stk**.

The argument *anInteger*, if provided, indicates how many contexts of the stack to display, starting with the current active context.

```
topaz 1> where
1 System class >> signal:args:signalDictionary: @8 line 15
==> 2 Object >> _error:args: @7 line 10
3 Object >> _error: @2 line 5
4 Number >> _errorDivideByZero @1 line 5
5 SmallInteger >> / @4 line 6
6 Executed Code @2 line 1
topaz 1> where 3
2 Object >> _error:args: @7 line 10
3 Object >> _error: @2 line 5
4 Number >> _errorDivideByZero @1 line 5
```

—
|

Topaz Command-Line Syntax

When Topaz is invoked with the **-l** option, it initiates the program with a linked, as opposed to a remote (RPC) session. Other command-line options give additional control. This section presents the formal command syntax followed by a complete list of command-line options.

A.1 Command-Line Syntax

By default the **topaz** command invokes an RPC executable. This is the same as specifying the **-r** option on the topaz command line:

```
topaz [ -r ] [ -n netLdiName ] [ -q ] [-I topazini] [ -i ]  
      [ -h ]
```

When invoked with the **-l** option, Topaz runs in linked mode. The command line accepts some additional options for the linked version:

```
topaz -l [ -n netLdiName ] [ -e exeConfig ] [ -z systemConfig ]  
      [-T tocSizeKB] [ -q ] [-I topazini] [ -i ] [ -h ]
```

A.2 Options

Arguments are optional. Other than the `-l` option to specify linked topaz, these arguments may not be needed for a standard GemStone configuration.

-l Invoke the linked version of Topaz. In this version, Topaz and Gem (the GemStone session) exist as a single process, which significantly enhances performance. The linked version can run only one linked session. Any additional sessions are initiated as remote procedure call sessions. If you don't specify this parameter, the remote procedure call version of Topaz is invoked.

If subsequent login parameters set `gemnetid` to a version of `gemnetobject`, topaz will login RPC rather than linked, regardless of the `-l` option.

-r Invoke the remote procedure call version of Topaz. In this version, Gems exist as separate processes. If you intend to run multiple GemStone sessions simultaneously, or if you will be running Topaz and your GemStone session on separate nodes, then you must use this version. If you don't specify `-l` or `-r`, Topaz defaults to the remote procedure call version.

-n *netLdiName*

The name of the network server process. This may be specified as a GemStone network resource specification. If you don't explicitly specify this parameter, Topaz will look for

- (1) a name specified by the `GEMSTONE_NRS_ALL` environment variable
- (2) a GemStone network server named `gs64ldi`

-e *exeConfig*

Executable-specific configuration file. If this argument is not present, the Topaz command uses the customary `GEMSTONE_EXE_CONF` search sequence described in the "Configuration Files" chapter of your *GemStone/S 64 Bit System Administration Guide*. Only applies to linked sessions.

-z *systemConfig*

System configuration file. If this argument is not present, the topaz command uses the customary `GEMSTONE_SYS_CONF` search sequence described in the "Configuration Files" chapter of your *GemStone/S 64 Bit System Administration Guide*. Only applies to linked sessions.

-T *tocSizeKB*

Sets the `GEM_TEMPOBJ_CACHE_SIZE` that will be used. This overrides any settings provided in configuration files passed as arguments with the `-e` or `-z` options. Only applies to linked sessions.

-i Ignore the topaz startup file `.topazini`.

- I** *topazini*
Specify a complete path and file to a topazini initialization files, and use this rather than any *.topazini* in the default location.
- q** Start Topaz in quiet mode, suppressing printout of the banner and other information.
- h** Print a usage line and exit.

—
|

Network Resource String Syntax

This appendix describes the syntax for network resource strings. A network resource string (NRS) provides a means for uniquely identifying a GemStone file or process by specifying its location on the network, its type, and authorization information. GemStone utilities use network resource strings to request services from a NetLDI.

B.1 Overview

One common application of NRS strings is the specification of login parameters for a remote process (RPC) GemStone application. An RPC login typically requires you to specify a GemStone repository monitor and a Gem service on a remote server, using NRS strings that include the remote server's hostname. For example, to log in from Topaz to a Stone process called "gs64stone" running on node "handel", you would specify two NRS strings:

```
topaz> set gemstone !@handel!gs64stone
topaz> set gemnetid !@handel!gemnetobject
```

Many GemStone processes use network resource strings, so the strings show up in places where command arguments are recorded, such as the GemStone log file. Looking at log messages will show you the way an NRS works. For example:

```
Opening transaction log file for read,  
filename = !tcp@oboe#dbf!/user1/gemstone/data/tranlog0.dbf
```

An NRS can contain spaces and special characters. On heterogeneous network systems, you need to keep in mind that the various UNIX shells have their own rules for interpreting these characters. If you have a problem getting a command to work with an NRS as part of the command line, check the syntax of the NRS recorded in the log file. It may be that the shell didn't expand the string as you expected.

NOTE

Before you begin using network resource strings, make sure you understand the behavior of the software that will process the command.

See each operating system's documentation for a full discussion of its own rules about escaping certain characters in NRS strings that are entered at a command prompt.

If there is a space in the NRS, you can replace the space with a colon (:), or you can enclose the string in quotes (" "). For example, the following network resource strings are equivalent:

```
% waitstone !tcp@oboe#auth:user@password!gs64stone  
% waitstone "!tcp@oboe#auth user@password!gs64stone"
```

B.2 Defaults

The following items uniquely identify a network resource:

- communications protocol— such as TCP/IP
- destination node—the host that has the resource
- authentication of the user—such as a system authorization code
- resource type—such as server, database extent, or task
- environment—such as a NetLDI, a directory, or the name of a log file
- resource name—the name of the specific resource being requested.

A network resource string can include some or all of this information. In most cases, you need not fill in all of the fields in a network resource string. The

information required depends upon the nature of the utility being executed and the task to be accomplished. Most GemStone utilities provide some context-sensitive defaults. For example, the Topaz interface prefixes the name of a Stone process with the **#server** resource identifier.

When a utility needs a value for which it does not have a built-in default, it relies on the system-wide defaults described in the syntax productions in “Syntax” on page 150. You can supply your own default values for NRS modifiers by defining an environment variable named GEMSTONE_NRS_ALL in the form of the *nrs-header* production described in the Syntax section. If GEMSTONE_NRS_ALL defines a value for the desired field, that value is used in place of the system default. (There can be no meaningful default value for “resource name.”)

A GemStone utility picks up the value of GEMSTONE_NRS_ALL as it is defined when the utility is started. Subsequent changes to the environment variable are not reflected in the behavior of an already-running utility.

When a client utility submits a request to a NetLDI, the utility uses its own defaults and those gleaned from its environment to build the NRS. After the NRS is submitted to it, the NetLDI then applies additional defaults if needed. Values submitted by the client utility take precedence over those provided by the NetLDI.

B.3 Notation

Terminal symbols are printed in boldface. They appear in a network resource string as written:

#server

Nonterminal symbols are printed in italics. They are defined in terms of terminal symbols and other nonterminal symbols:

username ::= nrs-identifier

Items enclosed in square brackets are optional. When they appear, they can appear only one time:

address-modifier ::= [protocol] [@ node]

Items enclosed in curly braces are also optional. When they appear, they can appear more than once:

nrs-header ::= ! [address-modifier] {keyword-modifier} !

Parentheses and vertical bars denote multiple options. Any single item on the list can be chosen:

protocol ::= (**tcp** | **serial** | **default**)

B.4 Syntax

nrs ::= [*nrs-header*] *nrs-body*

where:

nrs-header ::= ! [*address-modifier*] {*keyword-modifier*} [*resource-modifier*]!

All modifiers are optional, and defaults apply if a modifier is omitted. The value of an environment variable can be placed in an NRS by preceding the name of the variable with "\$". If the name needs to be followed by alphanumeric text, then it can be bracketed by "{" and "}". If an environment variable named `foo` exists, then either of the following will cause it to be expanded: `$foo` or `${foo}`. Environment variables are only expanded in the *nrs-header*. The *nrs-body* is never parsed.

address-modifier ::= [*protocol*] [@ *node*]

Specifies where the network resource is.

protocol ::= (**tcp** | **serial** | **default**)

Supports heterogeneous connections by predicating address on a network type. If no protocol is specified, `GCI_NET_DEFAULT_PROTOCOL` is used. On UNIX hosts, this default is **tcp**.

node ::= *nrs-identifier*

If no node is specified, the current machine's network node name is used. The identifier may also be an Internet-style numeric address. For example:

```
!tcp@120.0.0.4#server!cornerstone
```

nrs-identifier ::= *identifier*

Identifiers are runs of characters; the special characters `!`, `#`, `$`, `@`, `^` and white space (blank, tab, newline) must be preceded by a "^". Identifiers are words in the UNIX sense.

keyword-modifier ::= (*authorization-modifier* | *environment-modifier*)

Keyword modifiers may be given in any order. If a keyword modifier is specified more than once, the latter replaces the former. If a keyword modifier takes an argument, then the keyword may be separated from the argument by a space or a colon.

authorization-modifier ::= ((**#auth** | **#encrypted**) [:] *username* [@ *password*])

#auth specifies a valid user on the target network. A valid password is needed only if the resource type requires authentication. **#encrypted** is used by

GemStone utilities. If no authentication information is specified, the system will try to get it from the `.netrc` file. This type of authorization is the default.

username ::= nrs-identifier

If no user name is specified, the default is the current user.
(See the earlier discussion of *nrs-identifier*.)

password ::= nrs-identifier

If no password is specified, the system will try to obtain it from the user's `.netrc` file. (See the earlier discussion of *nrs-identifier*.)

environment-modifier ::= (#netldi | #dir | #log) [:] nrs-identifier

#netldi causes the named NetLDI to be used to service the request. If no NetLDI is specified, the default is `gs64ldi`. When you specify the **#netldi** option, the *nrs-identifier* is either the name of a NetLDI service or the port number at which a NetLDI is running.

#dir sets the default directory of the network resource. It has no effect if the resource already exists. If a directory is not set, the pattern “%H” (defined below) is used. (See the earlier discussion of *nrs-identifier*.)

#log sets the name of the log file of the network resource. It has no effect if the resource already exists. If the log name is a relative path, it is relative to the working directory. If a log name is not set, the pattern “%N%P%M.log” (defined below) is used. (See the earlier discussion of *nrs-identifier*.)

The argument to **#dir** or **#log** can contain patterns that are expanded in the context of the created resource. The following patterns are supported:

%H	home directory
%M	machine's network node name
%N	executable's base name
%P	process pid
%U	user name
%%	%

resource-modifier ::= (#server | #spawn | #task | #dbf | #monitor | #file)

Identifies the intended purpose of the string in the *nrs-body*. An NRS can contain only one resource modifier. The default resource modifier is context sensitive. For instance, if the system expects an NRS for a database file, then the default is **#dbf**.

#server directs the NetLDI to search for the network address of a server, such as a Stone or another NetLDI. If successful, it returns the address. The *nrs-body* is a network server name. A successful lookup means only that the service has been defined; it does not indicate whether the service is currently running. A

new process will not be started. (Authorization is needed only if the NetLDI is on a remote node and is running in secure mode.)

#task starts a new Gem. The *nrs-body* is a NetLDI service name (such as “gemnetobject”), followed by arguments to the command line. The NetLDI creates the named service by looking first for an entry in `$GEMSTONE/bin/services.dat`, and then in the user’s home directory for an executable having that name. The NetLDI returns the network address of the service. (Authorization is needed to create a new process unless the NetLDI is in guest mode.) The **#task** resource modifier is also used internally to create page servers.

#dbf is used to access a database file. The *nrs-body* is the file spec of a GemStone database file. The NetLDI creates a page server on the given node to access the database and returns the network address of the page server. (Authorization is needed unless the NetLDI is in guest mode).

#spawn is used internally to start the garbage collection and other service Gem processes.

#monitor is used internally to start up a shared page cache monitor.

#file means the *nrs-body* is the file spec of a file on the given host (not currently implemented).

nrs-body ::= unformatted text, to end of string

The *nrs-body* is interpreted according to the context established by the *resource-modifier*. No extended identifier expansion is done in the *nrs-body*, and no special escapes are needed.

Symbols

\$ (for character literal) 41
@ (for OOP literal) 41, 61
^ (current class) 41, 61
! (remark) 116
** (last result) 41, 61

A

abort command 33, 59
aborting transactions 32
automatic batch processing 36

B

batch processing from an input file 36
begin command 60

break command 61
classmethod 61
clear 51, 62
clear all 51, 62
display 50, 62
message 49
method 49, 61
breakpoints 48, 61
 and special methods 61
 clearing 51, 62
 continuing GemStone Smalltalk execution
 after 67
 deleting 51
 listing 50, 62, 97
 method 49, 61
 methods that cannot have 61
 setting 49, 61
byte objects
 limiting display of 28, 94
 storing into with structural access 106
 structural access 107

byte values, displaying 28, 70, 108

C

`_cacheName`

System method 121

`CacheName` (status output) 121

call stack

and GemStone Smalltalk debugging 128

displaying contents of active 128

examining 128

and GemStone Smalltalk debugging 128

redefining 131

removing 132

category

current 30, 31

listing 32

setting the current 121

category command 64

characters, Topaz syntax for 41

class

creating with **set class** command 121

current 30, 31

filing out 34

modifying with **set class** command 121

setting the current 121

class instances

Topaz syntax for 79, 82

class methods

changing 31

compiling 65

creating 31, 75

editing 31

modifying 75

classmethod command 31, 65

command-line syntax 143

commands

abbreviation of 57

abort 33, 59

begin 60

break 61

case-sensitivity of 57

category 64

classmethod 65

commit 32, 66

continue 52, 67

define 43, 68

disassem 69

display 28, 70, 108

doit 72

down 73

edit 74

errorcount 76

exit 46, 77, 114

expectbug 78

expecterror 79

expectvalue 82

fileout 34, 84

frame 85

gcitrace 86

help 87

iferr 88

iferr_clear 89

iferr_list 90

iferror 91

input 92

level 26, 93

limit 28, 94

list 95

login 99

logout 45, 101

lookup 102

method 31, 103

nbresult 105

nbrun 104

object 106

omit 28, 70, 108

opal—see **doit**

output 110

printit 24, 112

protectmethods 113

quit 114

releaseall 115

remark 116

removeallclassmethods 118

removeallmethods 117, 118
run—see **printit**
send 45, 120
set 19, 121
shell 126
spawn 127
stack 128
status 23, 31, 133
step 48, 134
stk 135
 syntax of 57
temporary 53, 136
time 138
unprotectmethods 139
up 140
where 141
 comments 116
commit command 32, 66
 committing transactions 32
 context

- and GemStone Smalltalk debugging 128
- displaying the active 130
- listing method breakpoints 50, 97
- listing step points in 96
- redefining the active 54, 130
- selecting 54

continue command 52, 67
 Control-C handling 37
 current category, setting 64, 121
 current class

- and **classmethod** command 65
- and **method** command 103
- setting 121

 current time, displaying 138

D

debugging 47–51, 128, 134, 135, 136
 and execution context 54, 128
define command 43, 68
disassem command 69
display command 28, 70, 108
oops and stack display 128

display level 26–27
 maximum 93
 display of results, controlling 25
doit command 72
down command 73

E

edit command 74
 and **set editorname** command 122
classmethod 31, 75
last 25, 74
method 31, 75
new classmethod 32, 75
new method 32, 74
new text 25
 used with **set** command 121
 editing GemStone Smalltalk expressions 25
 error status 77, 114
errorcount command 76
 errors, continuing GemStone Smalltalk
 execution after 67
 execution, stepping through 48, 96
exit command 46, 77, 114
expectbug command 78
expecterror command 79
expectvalue command 82

F

file

- appending to 33
- input 36, 37
- output 33, 37
- redirection 33

fileout command 34, 84
 finding method in hierarchy 102
 Floats, Topaz syntax for 41
frame command 85
 ftplogin. 23, 124

G

gcitrace command 86
gemnetobjcsh 122
gemnetobject 122
GemStone
 aborting a transaction 59
 call stack and GemStone Smalltalk
 debugging 128
 committing a transaction 66
 context and error handling 128
 examining the call stack 128
 interrupting 37
 logging in 18
 logging out 45
 multiple sessions 38, 124
 service, setting 122
GemStone name 18, 123
 setting 99
GemStone password 18
 setting 99, 124
GemStone service name, setting 99
GemStone Smalltalk
 breakpoints 61, 97
 continuing execution 52, 67
 debugging 47–51, 128, 134, 135, 136
 editing expressions 25
 editing source code 74, 122
 executing expressions 24, 72, 112
 sending messages 120
GemStone username 18
 setting 99, 125

H

help command 23, 87
 hexadecimal values, displaying 28, 70, 108
 host password 18, 123
 setting 99
 host username 18, 123

I

iferr command 88
iferr_clear command 89
iferr_list command 90
iferror command 91
 initialization file
 and **set host password** command 123
 and **set password** command 124
 used to set login parameters 99
input command 37, 92
 pop 92
 instance methods
 compiling 103
 creating 74
 modifying 75
 instance variables
 displaying 26, 28, 70, 93, 108
 returning the values of 106
 instances of a class, Topaz syntax for 79, 82
 integers, Topaz syntax for 41
 interrupting execution 37

L

level command 26, 93
limit command 28, 94
 bytes 94
 oops 94
list command 95
 breaks 50, 97
 breaks classmethod 97
 breaks method 97
 classmethod 32
 method 32
 steps 96
 steps classmethod 96
 steps method 48, 96
 logging a session 37
 logging in to GemStone 18, 99
login command 99
 login initialization file 21

login parameters 18–23
and **set** command 121
displaying the value of 133

logout command 45, 101

lookup command 102

M

message breakpoints
listing 50, 62
setting 49

method breakpoints 49, 61
listing 50, 62, 97
setting 49, 61

method command 31, 103

method compilations and **set category**
command 121

methods

- compilation 103
- compilation and current category 64
- creating 30, 74, 122
- editing 31
- examining and modifying arguments 53
- filing out 34
- finding in hierarchy 102
- listing 32, 102
- modifying 30
- stepping through execution 96, 134

multiple sessions 38, 124

N

nbresult command 105

nbrun command 104

.netrc 23, 124

network

- resource string syntax 147

network communications and login
parameters 99

network initialization file 23, 124

network resource string (NRS) 99

network server process, establishing the name
of 122

nonsequenceable collections (NSCs)
structural access 106

NRS (network resource string)
syntax 147

O

object command 106

- at:** 40, 106

- at:put:** 106, 107

object headers 30

objects, syntax for specifying 41

omit command 28, 70, 108

- oops**, and stack display 51, 128

OOPs

- displaying 28, 70, 108

- limiting display of 94

- Topaz syntax for 41, 61

opal command—see **doit**

operating system error status 77, 114

output command 110

- and host environment names 110

- pop** 110

- push** 33, 110

output to a file 33

P

password

- GemStone 18, 124

- host 18, 123

pause message, continuing GemStone
Smalltalk execution after 67

printit command 24, 112

- editing the text of 74

protectmethods command 113

Q

quit command 114

quitting Topaz 46

R

recording a session 37
releaseall command 115
remark command 116
removeallclassmethods command 118
removeallmethods command 117, 118
run command—see **printit**

S

send command 45, 120
service name, GemStone 122
session numbers 38, 124
sessions, multiple 38, 124
set command 19, 121

- cachename** 121
- category** 121
- class** 30, 121
 - and **edit classmethod** command 75
 - and **edit method** command 75
 - and **edit new classmethod** command 75
 - and **edit new method** command 74
- editorname** 25, 74, 122
- establishing login parameters 99, 121
- gemnetid** 122
- gemstone** 123
- hostpassword** 123
- hostusername** 123
- password** 124
- session** 38, 124
 - and local variables 68
- sourcestringclass** 125
- username** 125

shell command 126
spawn command 127
special methods

- and breakpoints 61

stack command 128

- all** 54, 131
- change** 54, 131
- delete** 132
- nosave** 131
- save** 131
- scope** 54, 130

stack, redefining the active 54
standard input, redirecting 37
standard output, redirecting 33
status command 23, 31, 133
stdin 33
stdout 33
step command 48, 134

- into** 53, 134
- over** 52, 134

step points 48

- examining 48, 96
- methods that have no 61

stk command 135
stopping execution 37
Strings

- limiting display of 28, 94
- Topaz syntax for 41

structural access 39–106

- and **object** command 106

Symbols, Topaz syntax for 41

T

temporaries, examining and modifying 53, 136
temporary command 53, 136
time command 138
Topaz

- command-line syntax 143
- exiting 46
- initialization file 21
- interrupting 37
- invoking 16
- linked version 17
- redirecting input 92
- redirecting output 110

- RPC version 17
- syntax for characters 41
- syntax for commands 57
- syntax for Floats 41
- syntax for instances of a class 79, 82
- syntax for integers 41
- syntax for literals 41
- syntax for OOPs 41, 61
- syntax for Strings 41
- syntax for Symbols 41
- syntax for variable names 41, 61

- Topaz initialization file
 - and **set host password** command 123
 - and **set password** command 124

- Topaz variables
 - and **define** command 68
 - and **object** command 106

- topaz.ini 21
- topazini.tpz 21
- transactions, aborting 33, 59
- transactions, committing 33

U

- unprotectmethods** command 139
- up** command 140
- username
 - GemStone 18, 125
 - host 18, 123

V

- variable names, Topaz syntax for 41, 61
- variables, local 42
 - and **define** command 68
 - and **object** command 106
- clearing definition of 44

- variables, predefined
 - CurrentCategory 44
 - CurrentClass 44
 - ErrorContext 44
 - LastResult 44
 - LastText 44, 74
 - myUserProfile 44

W

- where** command 141
- writing to a file 84

—
|