Cincom®

Cincom Smalltalk™

Internet Client Developer's Guide

P46-0134-07

SIMPLIFICATION THROUGH INNOVATION®

Part Number: P46-0134-07

Software Release 7.6

Cincom Systems, Inc.

55 Merchant Street

Cincinnati, Ohio 45246

Phone: (513) 612-2300

Fax: (513) 612-2000

World Wide Web: http://www.cincom.com

Contents

Chapter 2 FTP

Chapter 4 HTTP

About This Book

Introduction

This document describes the VisualWorks Net Clients libraries and frameworks for building internet client applications. Standard internet client services include FTP, HTML, and E-mail.

These services are built on top of various underlying technologies. For client-server communications, these are all built on top of an underlying Socket communications technology, which is discussed in *Basic Libraries*. HTML and email use Internet Message and MIME formats, the data structures that are exchanged over the internet, which are described in Chapter 3, "Internet Messages and MIME Types."

Net Clients is an integral part of the VisualWorks technologies that enable you to build applications to take advantage of the internet and e-business.

In addition to Net Clients, VisualWorks includes:

- Security support for common symmetric and public key security algorithms (refer to the *Security Guide*)
- WebServices, for XML-based web service providing protocols, such as SOAP and WSDL (refer to the *Web Service Developer's Guide*)
- WebToolkit and VisualWave, for developing Web server based applications (refer to the *Web Application Developer's Guide*)
- Distributed Smalltalk, for distributed applications, including CORBA complient applications (refer to the *Distributed Smalltalk Application Developer's Guide*)
- Opentalk, for general communications protocol support (refer to the *Opentalk Protocol Layer Developer's Guide*).

Audience

This document is intended for new and experienced developers to quickly become productive developing internet client applications using the Net Clients capabilities of VisualWorks.

It is assumed that you have a beginning knowledge of programming in a Smalltalk environment, though not necessarily with VisualWorks. For introductory level information, the on-line *VisualWorks Tutorial* (available at http://www.cincom.com/smalltalk/tutorial), and the *Application Developer's Guide*.

Organization

This document begins with a general overview of the Net Clients libraries and frameworks, including some general purpose classes and tools that are used throughout the specific components.

Following an introduction to socket programming in VisualWorks, the chapters proceed through the higher-level protocols used for internet communications and how to employ them in VisualWorks.

The chapters are as follows:

Chapter 1, "Introduction to Net Clients" provides an overview of the protocols and other features supported by Net Clients. It describes loading the Net Client libraries, how to access these libraries in your application, and a few tools provided by Net Clients.

Chapter 2, "FTP" covers the VisualWorks implementation of the common File Transfer Protocol for clients. FTP is a very simple protocol for transferring files between hosts using a TCP/IP connection.

Chapter 3, "Internet Messages and MIME Types" describes building internet messages out of mime-type entities, and accessing the parts of messages. Messages are used for HTTP and email services.

Chapter 4, "HTTP" sending requests and receiving responses from an HTTP server. Requests and responses are in the form of internet messages, as described in the previous chapter.

Chapter 5, "Email" describes building email clients to send and receive mail messages. The three major protocols, POP3 and IMAP for receiving, SMTP for sending, are covered.

For a description of the basic VisualWorks XML framework, refer to the *Application Developer's Guide*. For XML-to-Smalltalk mapping, refer to the *Web Service Developer's Guide*.

Conventions

We have followed a variety of conventions, which are standard in the VisualWorks documentation.

Typographic Conventions

The following fonts are used to indicate special terms:

Example	Description
template	Indicates new terms where they are defined, emphasized words, book titles, and words as words.
`cover.doc`	Indicates filenames, pathnames, commands, and other constructs to be entered outside VisualWorks (for example, at a command line).
`filename.xwd`	Indicates a variable element for which you must substitute a value.
windowSpec	Indicates Smalltalk constructs; it also indicates any other information that you enter through the VisualWorks graphical user interface.
Edit menu	Indicates VisualWorks user-interface labels for menu names, dialog-box fields, and buttons; it also indicates emphasis in Smalltalk code samples.

Special Symbols

This book uses the following symbols to designate certain items or relationships:

Examples	Description
File ✝ **New**	Indicates the name of an item (New) on a menu (File).
<Return> key <Select> button <Operate> menu	Indicates the name of a keyboard key or mouse button; it also indicates the pop-up menu that is displayed by pressing the mouse button of the same name.
<Control>-<g>	Indicates two keys that must be pressed simultaneously.
<Escape> <c>	Indicates two keys that must be pressed sequentially.
Integer>>asCharacter	Indicates an instance method defined in a class.
Float class>>pi	Indicates a class method defined in a class.

Mouse Buttons and Menus

VisualWorks supports a one-, two-, or three-button mouse common on various platforms. Smalltalk traditionally expects a three-button mouse, where the buttons are denoted by the logical names <Select>, <Operate>, and <Window>:

<Select> button	*Select* (or choose) a window location or a menu item, position the text cursor, or highlight text.
<Operate> button	Bring up a menu of *operations* that are appropriate for the current view or selection. The menu that is displayed is referred to as the *<Operate> menu*.
<Window> button	Bring up the menu of actions that can be performed on any VisualWorks *window* (except dialogs), such as **move** and **close**. The menu that is displayed is referred to as the *<Window> menu*.

These buttons correspond to the following mouse buttons or combinations:

	3-Button	2-Button	1-Button
<Select>	Left button	Left button	Button
<Operate>	Right button	Right button	<Option>+<Select>
<Window>	Middle button	<Ctrl> + <Select>	<Command>+<Select>

Getting Help

There are many sources of technical help available to users of VisualWorks. Cincom technical support options are available to users who have purchased a commercial license. Public support options are available to both commercial and non-commercial license holders.

Commercial Licensees

If, after reading the documentation, you find that you need additional help, you can contact Cincom Technical Support. Cincom provides all customers with help on product installation. For other problems there are several service plans available. For more information, send email to supportweb@cincom.com.

Before Contacting Technical Support

When you need to contact a technical support representative, please be prepared to provide the following information:

- The *version id,* which indicates the version of the product you are using. Choose **Help ✝ About VisualWorks** in the VisualWorks main window. The version number can be found in the resulting dialog under **Version Id:**.
- Any modifications (*patch files*) distributed by Cincom that you have imported into the standard image. Choose **Help ✝ About VisualWorks** in the VisualWorks main window. All installed patches can be found in the resulting dialog under **Patches:**.
- The complete error message and stack trace, if an error notifier is the symptom of the problem. To do so, select **copy stack** in the error notifier window (or in the stack view of the spawned Debugger). Then paste the text into a file that you can send to technical support.

Contacting Technical Support

Cincom Technical Support provides assistance by:

Electronic Mail

To get technical assistance on VisualWorks products, send email to:

supportweb@cincom.com.

Web

In addition to product and company information, technical support information is available on the Cincom website:

http://supportweb.cincom.com

Telephone

Within North America, you can call Cincom Technical Support at (800) 727-3525. Operating hours are Monday through Friday from 8:30 a.m. to 5:00 p.m., Eastern time.

Outside North America, you must contact the local authorized reseller of Cincom products to find out the telephone numbers and hours for technical support.

Non-Commercial Licensees

VisualWorks Non-Commercial is provided “as is,” without any technical support from Cincom. There are, however, on-line sources of help available on VisualWorks and its add-on components. Be assured, you are *not* alone. Many of these resources are valuable to commercial licensees as well.

The University of Illinois at Urbana-Champaign very kindly provides several resources on VisualWorks and Smalltalk:

- A mailing list for users of VisualWorks Non-Commercial, which serves a growing community of VisualWorks Non-Commercial users. To subscribe or unsubscribe, send a message to:

  vwnc-request@cs.uiuc.edu

  with the SUBJECT of "subscribe" or "unsubscribe".
- An excellent Smalltalk archive is maintained by faculty and students at UIUC, who are long-time Smalltalk users and leading lights in the Smalltalk community, at:

  http://st-www.cs.uiuc.edu/
- A Wiki (a user-editable web site) for discussing any and all things VisualWorks related at:

  http://wiki.cs.uiuc.edu/VisualWorks
- A variety of tutorials and other materials specifically on VisualWorks at:

  http://wiki.cs.uiuc.edu/VisualWorks/Tutorials+and+courses

The Usenet Smalltalk news group, comp.lang.smalltalk, carries on active discussions about Smalltalk and VisualWorks, and is a good source for advice.

Additional Sources of Information

This is but one manual in the VisualWorks library. The Cincom Smalltalk publications website:

http://www.cincomsmalltalk.com/documentation/

is a resource for the most up to date versions of VisualWorks manuals and additional information pertaining to Cincom Smalltalk.

1

Introduction to Net Clients

Net Clients contains the frameworks and class libraries for VisualWorks that provide general access to common internet communications facilities. Specifically, it provides the following frameworks:

- **Uniform Resource Identification (URI) framework.** Represent and access resources.
- **FTP framework.** Access remote files and directories.
- **MIME-type framework.** Parsing and composing with e-mail/HTTP messages.
- **HTTP client engine framework.** Send and receive HTTP messages.
- **Mail frameworks:**
    - **POP3 client engine framework.** Receive e-mail messages and maintain mail folders.
    - **IMAP4rev1 client engine framework.** Receive e-mail messages and maintain mail folders.
    - **SMTP client engine framework.** Send e-mail messages.

The underlying technology for all of this is socket communications (BSD Sockets). Documentation for using sockets is included in this document, because their most common use is for internet communications.

In addition to the frameworks, Net Clients provides tools to assist in developing internet client applications:

- **Logging tool.** Records all outgoing and incoming messages for selected protocols.
- **Network Settings tool.** Specifies a default settings required by some protocols.

Loading Net Clients

Net Clients support is contained in a set of parcels. To access and use the code, you must load one or more of these parcels.

In the Settings Tool, there are options to autoload FTP, HTTP, and HTTPS parcels if their functionality is required. Set these options on the **URI** page of the Settings Tool.

To load the entire collection, load the single NetClients parcel. This one parcel loads all the others.

To better control your image file size, load only the components you intend to use in your application. The top-level component parcels, which are installed in the **`net/`** directory, are:

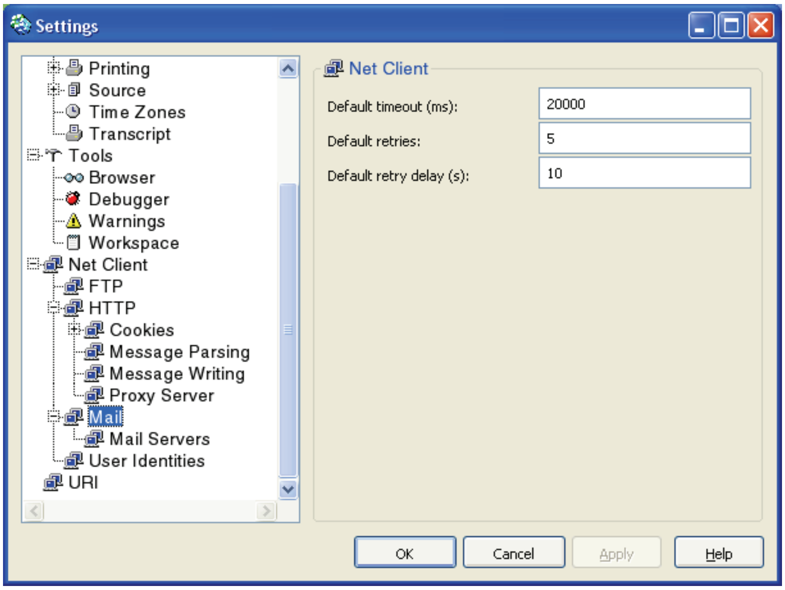
NetClients	Loads all NetClients support
FTPSupport	FTP support
HTTP	HTTP and HTTPS support
IMAP	IMAP support
Mail	Basic mail support
MIME	MIME support
NetConfigTool	NetClients settings tool
POP3	POP3 support
POP3S	Secure POP3 over SSL addition to POP3
SMTP	SMTP support
SMTPS	Secure SMTP over SLL addition to SMTP
URISupport	URI support
WebSupport	Web client support extensions
NetClientBase	Prerequisites for other parcels; automatically loaded

NetClient Settings

Settings Tool Pages

The parcel NetConfigTool installs a Net Client settings pages in the Settings Tool (**System ✝ Settings** in the Visual Launcher) to configure a few default values that are useful while working with Net Clients frameworks.

The tool has several pages providing relevant settings. To get information on a specific page, click Help with that page selected.



Default timeout (ms): 20000
Default retries: 5
Default retry delay (s): 10

In general, this tool is not suitable for use in applications, and you will want to provide your own account management tools. There may be situations where it is useful, however, such as providing a default email account for an automated reporting feature.

Setting values used by various classes throughout the NetClients framework, and are described in the relevant sections.

Settings API

Besides specific parameters set for a protocol instance, it is frequently useful to set default values for certain parameters. The Settings tool pages, as mentioned, are not generally appropriate for use in an application. To allow you to set default values more generally, several API methods are provided as class methods. Most of the API is defined in classes specific to the protocol. The following methods provide general setting API defined in NetClients class:

defaultDelaySeconds: *aNumber*
: Sets *aNumber* seconds between attempts to connect to the server, allowing for the server to become not busy.

defaultRetries: *aNumber*
: Sets *aNumber* of retries before quitting with failure.

defaultTimeout: *aNumber*
: Sets *aNumber* of seconds before timing out.

Logging Tool

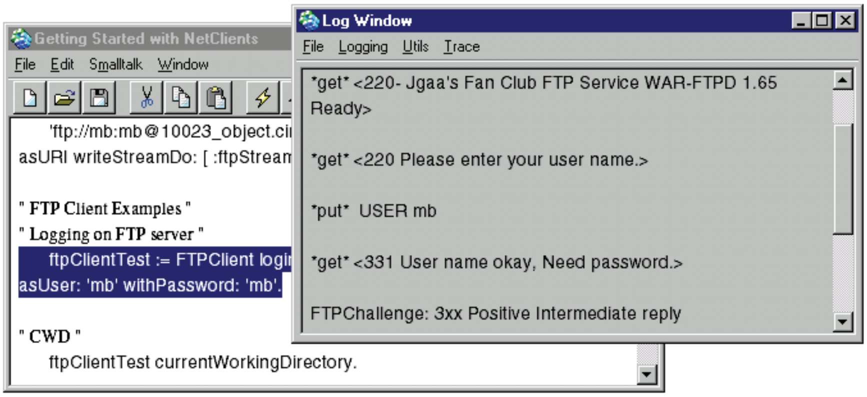
The Logging Tool is useful for understanding how network communications are happening, and especially for diagnosing problems with your network client code.

To use the logging tool, load the LoggingTool parcel (in the **`parcels/`** directory). Then open the tool by evaluating (DoIt) in a workspace:

```
LoggingTool open
```

NetClients components register themselves with the Logging Tool when they are loaded. The **Logging** menu lists all of the protocols currently registered. The submenus provide options for which messages to log. To add a message set, select it. Selected message sets are marked with a check mark.

To begin logging, select **Trace** ☩ **Trace ON**.



```
    'ftp://mb:mb@10023_object.ci
asURI writeStreamDo: [ :ftpStream

" FTP Client Examples "
" Logging on FTP server "
    ftpClientTest := FTPClient logi
asUser: 'mb' withPassword: 'mb'.

" CWD "
    ftpClientTest currentWorkingDirectory.
```

```
*get* <220- Jgaa's Fan Club FTP Service WAR-FTPD 1.65
Ready>

*get* <220 Please enter your user name.>

*put*  USER mb

*get* <331 User name okay, Need password.>

FTPChallenge: 3xx Positive Intermediate reply
```

Importing Net Clients into a Name Space

All Net Client support classes are defined in the Smalltalk.Net name space. Application code that uses these facilities should include this name space by a general import.

You can import the Net name space either into your own name space, making these resources available to every class in that name space, or into individual classes that need to use Net Client support.

To import the Net name space, include this line in the **imports:** line in either the class or the name space definition:

```
private Net.*
```

For example, you may have a network client application that uses many or all of the NetClient support classes. Your class definition, with the Net.* import, might look something like this:

```
Smalltalk.MyNamespace defineClass: #MyNetClientApp
    superclass: #{Core.Object}
    indexedType: #none
    private: false
    instanceVariableNames: 'user mailAddress proxy'
    classInstanceVariableNames: ''
    imports: 'private Net.*'
    category: 'Tools-Mail'
```

Because your application is defined in your own name space, and possibly only few of your classes require access to Net, it would be inappropriate to import Net into your name space. If, on the other hand, your network clients application were defined in its own sub-name space, then importing Net to that name space might be appropriate.

Common interface classes

There are a few classes that are invoked throughout the Net Clients framework: NetClient, NetUser, and URI.

NetClient

NetClient is the abstract superclass for the specific clients. However, a good deal of useful protocol, especially with respect to connections, is inherited from NetClient, and so you should be aware of it.

A useful set of methods is the instance creation class methods, which are used frequently in the examples.

connectToHost: *hostName*
connectToHost: *hostName* **port:** *portNum*
Creates a client instance and establishes a connection to *hostName*, a String, on the default port for the protocol. If *portNum* is specified, that port is used instead of the default.

host: *hostName*
host: *hostName* **port:** *portNum*
Creates a client instance targeted on *hostName*, a String, on the default port for the protocol. If *portNum* is specified, that port is used instead of the default.

Instance methods provide connection support, and allow you to set or get user and host information. Browse the **accessing** and **connection** method categories for useful messages.

NetUser

Most internet services require a user ID, and usually a password, to gain access (log in) to the service. For very simple services, such as FTP, the user name and password are simply String data.

Most of the clients, however, require the user ID and password to be specified by an instance of NetUser. A NetUser instance is a convenient holder for several pieces of user information, as is done by the Network Settings tool. It holds at least the user ID and password, but also the user's full name, an account name and an email address.

To create a NetUser instance, send a username:password: message to the class, with the user name and password specified as strings:

```
user := NetUser username: 'Fred' password: 'dont_ask'
```

To add other information items, send the appropriate set message:

```
user fullName: 'Frederick P. User';
    mailAddress: 'fpuser@someplace.net'.
```

There are also setter methods for changing the user name and password, as well as getter methods for all of these data items.

URI

Net Clients includes a framework for working with URIs (Uniform Resource Identifiers) and URLs (Universal Resource Locators). The framework provides an easy-to-use mechanism to create URL objects from a String, as well as a simple interface for performing some operations on the URL, such as reading and writing, if permitted by the resource.

A URI is a string that represents the address of a piece of information on the Internet. A URL is a type of URI that specifies the protocol type, such as FTP, HTTP, and MailTo.

Creating a URI

In VisualWorks, a URI is an object that is created by parsing a String specifying the URI. To parse the String, send it an asURI message:

```
'ftp://download.cincom.com/documents/public/some.doc' asURI
```

The result in this case is an FtpURL object, which was determined from the protocol prefix. If the prefix had been “http:” the result would be an HttpURL. If the protocol were an unknown type, such as “MailTo:”, the result would be an UnknownURLType.

The URI string has three main parts: the protocol, the host, and the path. The protocol is determined by reading from the beginning of the string to the colon (FTP). The host is read from the String, from immediately following the double slash (//) up to the first single slash (download.cincom.com). Everything else is the path.

For local files, you can generate a URI from a Filename:

```
'localfile.txt' asFilename asURI
```

A URI object can be returned to a String by sending asString to it.

Working with URI paths

Since URIs frequently have paths, as in the FTP URI above, it is useful to be able to travel the path, either up or down. Several messages are provided (implemented in URLwithPath) to do this.

The URI path is stored as an OrderedCollection of Strings, which are the string segments between slashes or following the last slash. So, in the FTP URI above, the path is stored as an OrderedCollection with 'documents', 'public', and 'some.doc'. The following path resolution messages operate on the path components. Browse URLwithPath for more methods.

construct: *pathString*

Returns a new URI differing from the receiver as specified by *pathString*.

resolveRelativePath: *pathString*

Returns a new URI resulting from removing the last path component and the applying *pathString*. Primarily useful when the last component is not a directory.

withComponent: *pathComponentString*

Returns a new URI with the *pathComponentString* appended. The *pathComponentString* is assumed to be URL-encoded (special characters are escaped).

withSimpleComponent: *pathComponentString*

Returns a new URI with the *pathComponentString* appended. The *pathComponentString* is assumed not to be URL-encoded (special characters are escaped), and so encodes them appropriately.

For example, given an URI to a file, you can create a new URI to a file relative to its path by sending a resolveRelativePath: message to the URI object. The argument is a String containing relative path information.

```
uri := 'ftp://download.cincom.com/documents/some.doc' asURI.
pix := uri resolveRelativePath: '../graphics/pic.jpg'.
```

Escaping characters in a URI

Certain characters must be “escaped” in an URI, that is represented by a percent sign (%) followed by their ASCII decimal value (e.g., '$' must be represented as '%24'). withSimpleComponent: escapes the necessary characters in a component before adding it. However, to escape the entire path of a URI you need to work on all components. Send an escaped: message to the URI with each component as argument and collect the results, then plug the path back in by sending a path: message:

```
| uri escPath |
uri := 'http://www.somesite.com/St&Mickey/buck$/index.html' asURI.
escPath := uri path collect: [ :comp | uri escaped: comp ].
uri path: escPath.
```

Operations on URIs

The URI framework also provides a simple way for a program to perform various operations on URI resource. FtpURL has the richest protocol, as described in “Using FtpURL” in Chapter 2, “FTP.”

Each type of URI represents a different accessing protocol, which has different limitations, so not all types of URI support the same accessing messages. These two messages for reading and writing data are common among the protocols:

readStreamDo: *aBlock*

Create a read stream on the URI resource and performs the read operations defined in *aBlock* on the stream.

writeStreamDo: *aBlock*

Creates a write stream on the URI resource, and performs the write operation defined in *aBlock* on the stream.

See the discussion of `FtpURL` for examples, and browse the other methods defined in the `FtpURL`, `FileURL`, and `HttpURL` classes.

2

FTP

Introduction

File Transfer Protocol is a simple protocol for copying files between host and client computers. Both systems must support TCP/IP .

FTP involves both client and server services. The FTP client requests a file transfer or other service from the FTP server. Most systems that have a TCP/IP protocol suite installed also have FTP.

VisualWorks FTP support currently implements client services only.

VisualWorks FTP connection services are built on top of the SocketAccessor TCP/IP services provided in the base VisualWorks, which allows FTP services to be added as a module.

The following basic operations are available with FTP:

- Copy a single or multiple files from one host to another
- List all accessible files on a target host
- Create and/or remove directories on a target host
- Identify the current directory on a target host
- Append a local file to a file located on a remote (target) host
- Append a file from one remote host to a file located on a second host

NetClients FTP Interfaces

NetClients provides two interface classes to FTP operations:

- **FtpURL** A simplified API that allows you to access remote files in a style similar to that provided by Filename for local files. See Using FtpURL.
- **FTPClient** The primary VisualWorks interface to FTP. See Using FTPClient.

In a few situations you may also need to issue “raw” FTP commands directly. FTPClient provides a way to do this as well. See Using FTP commands and responses.

The main API is provided by FTPClient, which typically “wraps” the command API and itself is defined in FTPClient. FtpURL is a simplified API that invokes FTPClient to perform the actual FTP transactions.

When to use FtpURL or FTPClient

Which of these interfaces you choose to use depends on a variety of factors. A rule of thumb is, if you are performing a very simple FTP operation, consider using FtpURL; otherwise, use FTPClient.

A major consideration is how many operations you are performing. FtpURL is useful to perform a single operation per connection, because its commands close the connection upon completion. So, if you only need to transfer a single file, and you can specify the file’s location as an FTP URI, you may be able to use FtpURL.

On the other hand, if you need to perform several operations using the same connection, such as getting a directory listing, changing directories, and then transferring a file, you need to use FTPClient. Using FTPClient you establish a connection and hold it open until you explicitly close it.

Another consideration is the amount of directory and file name management you need to perform. FtpURL operates on a URI, which is created by sending the asURI message to a string that specifies the transfer protocol, login information, and exact file locations. FTPClient commands, on the other hand, allow you a good deal more flexibility in managing the login procedure, selecting a directory, and specifying files.

Default Settings API

Two defaults settings for FTP sessions can be set either using the Settings tool or the following API methods. Both are class methods in the Net.Settings class.

ftpAnonymousLoginPassword: *aString*
By convention, the default password for anonymous FTP is the user's email address. For convenience, you may set a default address as a String.

ftpPassiveMode: *aBoolean*
If the client requires that the server be in passive mode, set the argument as true. The server is notified to enter passive mode.

FTP Basics

FTP Access and Security

To establish an FTP connection you need a user account (user name) and password on the remote server. Since FTP includes no facility for masking or encrypting passwords, logging in to a FTP connection carries some inherent risk if the transmission containing your password is intercepted. For this reason, it is best to use anonymous login whenever possible, or some other restricted access login.

Sites that make files available for public access usually support anonymous FTP, which allows you to log on using the **`anonymous`** user name and your email address as the password.

For sites that do not support anonymous FTP, or to access resources that are not available to the public, you need an account and password. To get an account, contact the system administrator.

Guarded and Unguarded Stream Transfers

Stream transfers are supported in two modes: guarded and unguarded.

Guarded transfers ensure that the connection is closed at the end of a transfer.

Unguarded transfers leave the connection open at the end of a transfer, ready for another operation. It is the programmer's responsibility to ensure that the connection is closed when all FTP operations are complete.

Passive and Active Modes

The standard way for an FTP session to begin is for the server to initiate the connection; this is active mode. Because a client may be behind a firewall that blocks the server from making the data connection, passive mode may be required. VisualWorks supports the ability to configure a server for passive data transfer.

In passive mode, the FTP server listens on the data port for a connection request from the client (the active process). When the connection is established, the data transfer begins between client and server, and the server sends a confirming reply to the client.

Using FtpURL

FtpURL provides a simplified interface for performing FTP transactions. It primarily provides utility methods allowing transactions to mimic Filename protocol. Using this interface, you can access remote files much the same way you access local files—by sending a message to a FtpURL specifying the remote file, rather than a file name.

FtpURL commands attempt to open a connection and, if successful, execute the command and then close the connection. Accordingly, this sort of FTP session is very short. For longer FTP sessions, executing a sequence of FTP commands, use FTPClient.

Identify a remote FTP file

A FtpURL object holds the information necessary to make an FTP connection, including:

- user name
- password
- host name
- port
- file path

The easiest way to create a FtpURL with this information is to send asURI to a String, in the form:

```
'ftp://<username>:<password>@<host>:<port>/<path>' asURI
```

Note that the file path separator character must be "/"; platform separators do not work at this time. Special characters, such as the "@" in an email address, must be replaced with a hexadecimal value (e.g., replace "@" with "%40"). See Special Symbols in the Access String below for replacement values).

Caution: Because passwords are not encrypted when logging onto a FTP host, specifying a user name and password may present a security risk. For this reason, it is preferable to use anonymous access (the default) whenever possible.

Defaults

The **`<port>`**, **`<username>`**, and **`<password>`** parameters are all optional. If not specified:

- the default port is 21
- the user name is **`anonymous`**
- you are prompted for the password. For anonymous FTP, this is usually your email address.

Using all of these defaults, the minimum FtpURL forming command will be in the form:

‘ftp://<host>/<path>’ asURI

Special Symbols in the Access String

If your access String includes certain symbols except where they are required by the String format (for example, an “@” in an email address), you must replace the symbol with a string representing its hex value (for example, replace "@" with "%40"). Other characters (especially in the password) should also be encoded so they are not confused with the URL syntax.

Some of the more common symbols are:

Symbol	Replacement string (hex value)
% *(percent*)	%25
, *(comma)*	%2C
/ *(forward slash)*	%2F
: *(colon)*	%3A
; *(semi-colon)*	%3A
@ *(at sign)*	%40
(space)	%20

FtpURL Exception Handling

FtpURL commands include a default exception handling mechanism.

Exception conditions that prevent the operation from being performed, such as inadequate permission to upload a file, display a notifier, including the FTP response number and a brief text description.

Transient conditions, such as a log-in challenge that can be handled by submitting an anonymous login, are handled transparently. A notifier is displayed only if the condition persists so the process cannot continue.

FtpURL invokes the exception handling mechanism by placing command processing code in a block that is passed to safelyExecuteBlock:, which in turn sends handleException. Both of these methods are defined in URLwithPath. To use the FtpURL API in an application, you may need to circumvent the default error handling described above, by overriding one or both of these methods in a subclass of FtpURL. For more on FTP exception handling, see "Handling FTP Exceptions" below.

Binary File Transfers

Binary mode is the default for file transfers using FtpURL. Transfers are performed by sending the copyTo: message to an FtpURL, with another FtpURL as argument. Accordingly, the same syntax is used for both uploads and downloads.

For text (ASCII) file transfers, use either the FtpURL stream methods (see Stream operations) or FTPClient commands (see Using FTPClient).

Download a file in binary mode

To download a file in binary mode, send copyTo: to an FtpURL for the remote, source file, with another FtpURL for the local, target directory or file as the message argument.

```
'ftp://anonymous:my%40email@yourftpserver/visualworks/welcome.pdf'
    asURI copyTo: 'welcome.pdf' asFilename asURI.
```

Upload a file in binary mode

To upload a file in binary mode, send copyTo: to an FtpURL for the local source file, with another FtpURL for the remote, target directory or file as the message argument.

```
'vwlogo.jpg' asFilename asURI copyTo:
    'ftp://anonymous:my%40email@yourftpserver/visualworks/vwlogo.jpg'
      asURI.
```

Directory operations

Create a new directory

To create a directory, send makeDirectory to the FtpURL.

```
'ftp://name:password@yourftpserver/visualworks/testDir2' asURI
    makeDirectory.
```

You must have suitable access to create a directory, which is seldom the case for anonymous FTP.

Delete a directory

To delete a directory, send removeDirectory to the FtpURL.

```
'ftp://name:password@yourftpserver/visualworks/testDir2' asURI
    removeDirectory.
```

You must have suitable access to delete a directory, which is seldom the case for anonymous FTP.

If the directory is not empty, an error notifier will display: “550 : The directory is not empty”. An error will also result if the directory cannot be found: “550 : The system cannot find the file specified”.

List files in a directory

To get a directory listing, send directoryContents to the FtpURL.

```
'ftp://name:password@yourftpserver/visualworks' asURI directoryContents.
```

This command, if successful, returns an OrderedCollection of FtpURLs, one for each item in the directory. Each FtpURL includes the user name and password, as specified in the command.

Operations on Files

Delete a file

To delete a file, send the delete message to the FtpURL. The login account must have suitable access privilege on the server.

```
'ftp://name:password@yourftpserver/visualworks/file1.txt' asURI delete.
```

An error notifier will display if the file cannot be found: “550 : The system cannot find the file specified”.

Determine the size of a file

To get the size (in bytes) of a file, send the fileSize message to the FtpURL.

```
'ftp://name:password@yourftpserver/visualworks/file1.txt' asURI fileSize.
```

Testing

Determine if a file or directory exists

To determine whether a file exists on the server, send the exists message to the FtpURL. The message returns either **true** or **false**, depending on the existence of the file or directory.

```
'ftp://name:password@yourftpserver/visualworks/file1.txt' asURI exists.
```

Determine if the URI is a directory

To determine whether a target is a directory, send the isDirectory message to the FtpURL. The message returns either **true** if the target is a directory, and **false** otherwise.

```
'ftp://name:password@yourftpserver/visualworks/file1.txt' asURI isDirectory.
```

Stream operations

Two stream methods are provided for performing text file transfers. Both of these methods use a guarded transfer, which ensures that streams are closed after use:

readStreamDo:
Read the resource that the URI represents. The message takes a 2-argument block as its argument, with the data stream as the first argument and a Dictionary of properties as the second argument. If the URI allows you to know or guess the MIME type of the data, that is included in the dictionary under the key #MIME. The dictionary may have other keys, depending on the type of URI and the server which holds the resource. The stream must be used within the block only, because it will be closed when the block finishes executing. The return value of readStreamDo: will be the value returned by the block.

writeStreamDo:
Write a new resource to the target URI. The argument to the message is a block that takes a stream as its only argument. The stream must be used within the block only. It will be closed when the block finishes executing. The return value of writeStreamDo: will be the value returned by the block.

For more information about stream operations, refer to the *VisualWorks Application Developer's Guide*.

Create a file on a remote server

To create a file, simply open a write stream on the remote server and write the file contents. The file may be empty, creating an empty file.

```
'ftp://user:password@yourftpserver/visualworks/file1.txt' asURI
    writeStreamDo: [ :ftpStream | ftpStream text;
        nextPutAll: 'blah blah'; cr].
```

Upload a file to a remote server

To upload a file to a server create a ReadStream on the local file and a WriteStream on a FtpURL for the remote file, then write the input data to the output stream. Stream data is assumed to be byte data, so is suitable for both text and binary files.

```
| input output |
input := 'test.text' asFilename readStream.
output := 'ftp://user:password@ftphost/visualworks/hello4.txt' asURI.
[input atEnd] whileFalse: [output writeStreamDo:
    [ :ftpStream | ftpStream text; nextPutAll: input text.]].
input close.
```

Note that writeStreamDo: on the FtpURL creates a guarded stream, which automatically closes the output stream when the transfer terminates. The input stream, which is simply opened on a file, still needs to be closed, as is done in the last line.

Download a file from a remote server (Text mode)

To download a file from a server, create a ReadStream on a FtpURL for the remote file, and a WriteStream on the local file, then write the input data to the output stream. Stream data is assumed to be byte data, so this method is suitable for both text and binary files.

```
| output |
output := 'helloworld.txt' asFilename writeStream.
'ftp://user:password@ftpserver/visualworks/hello.txt' asURI readStreamDo:
    [ :stream :params | stream text.[stream atEnd] whileFalse:
        [output nextPut: stream next] ].
output close.
```

Note that readStreamDo: on the FtpURL creates a guarded stream, which automatically closes the input stream when the transfer terminates. The output stream, which is simply opened on a file, still needs to be closed, as is done in the last line.

Read a file from a remote server

Instead of downloading a remote file to a local file, you can use FtpURL to read a stream for immediate processing by an application. In this example we simply inspect the results, but you can substitute other processing.

```
'ftp://user:password@ftpserver/visualworks/file1.txt' asURI
    readStreamDo: [:stream :params | stream contents inspect].
```

The read stream is a guarded stream (created by readStreamDo:), and so is closed automatically when the transfer ends.

Using FTPClient

FTPClient provides the main protocol for managing FTP sessions. FTPClient provides more complete and detailed control over an FTP session than does FtpURL, while relieving you of many of the details of controlling a session.

With this control comes added responsibility. Notably, you are responsible for explicitly closing the connection when the session is complete. Using an ensure: block is strongly recommended.

Using FTPClient is very similar to using FTP from a command-line interface. The sequence of messages sent for FTPClient closely mimics the typical sequence of commands used during an FTP session with a command-line interface.

FTPClient as an FTP session

An FTP session is initiated by opening a connection to a host and logging in. The connection remains open for processing commands until either the client or the host closes it.

In VisualWorks, an FTP session is represented by an instance of FTPClient. From a very high point of view, a session goes like this:

1. Create an FTPClient instance.
2. Connect the FTPClient to a host.
3. Log in to the host by sending a user name and password to the FTPClient.
4. Conduct various FTP operations by sending messages to the FTPClient.
5. Close the connection.

An overly simple example looks like this:

```
| ftpClient |

"create client"
    ftpClient := FTPClient new.
"connect to host"
    ftpClient connectToHost: 'ftp.parcplace.com' .
"log in"
    ftpClient login: 'anonymous' passwd: 'bruce@parcplace.com'.
"do stuff"
    Transcript nextPutAll: ftpClient currentWorkingDirectory displayString;
        cr ;
        flush.
"close connection"
    ftpClient close.
```

Enhancements are either elaborations on or simplifications of this sequence. For example, a simplification is that you can connect and log in using a single message. As an elaboration, you will want to do some exception handling and wrap your session in an ensure: block.

Connecting to an FTP host

FTPClient provides several instance creation methods that establish a connection and login to an FTP host.

connectToHost: *aHost*
connectToHost: *aHost* **port:** *aPort*
These methods open a connection to the specified host (host name or IP) and port (an integer). If no port is specified, the default (21) is used. Note that a user is not logged in by these commands; use FTPClient instance methods to send login information.

loginToHost: *aHostName* **asUser:** *userString* **withPassword:** *passwdString*
loginToHost: *aHostName* **asUser:** *userString* **withPassword:** *passwdString* **withAcct:** *acctString*
These methods open a connection to the specified host, and login using the specified user name and password. If an additional access account is required by the host, use the second form.

The connection remains open until you explicitly close it, or the connection is closed by the host. To close the connection, send a close message to the FTPClient. For catching an error resulting from the host closing the connection, see “Handling FTP Exceptions” below.

Using loginToHost:asUser:withPassword: the previous example simplifies slightly to:

```
| ftpClient |

"create client, connect, and log in"
    ftpClient := FTPClient loginToHost: 'ftp.parcplace.com'
        asUser: 'anonymous'
        withPassword: 'bruce@parcplace.com'.
"do stuff"
    Transcript nextPutAll: ftpClient currentWorkingDirectory displayString;
        cr ;
        flush.
"close connection"
    ftpClient close.
```

Re-establishing a connection

FTP connections can terminate for any number of reasons, and you may be left with an incomplete transaction. Most FTPClient methods automatically attempt to re-establish a connection, if necessary, in order to complete the requested operation.

If necessary, you can trap the FTPConnectionSignal and provide your own reconnection code. This exception is raised when an attempt is made to issue a command while the FTP control connection is closed. Refer to "Handling FTP Exceptions" below for more information on trapping exceptions.

On the client side, you can test to see if the connection is opened by asking the client's protocol interpreter whether it is closed. For example, if ftpClient is an FTPClient instance, you can send this message, which returns a Boolean:

```
ftpClient clientPI closed
```

Note that this message may return false, indicating that the protocol interpreter still believes the connection is open, even though the server may have closed the control connection.

If the connection is closed for either reason, you can reconnect by sending a reconnect message to the FTPClient instance:

```
ftpClient reconnect
```

Setting passive or active mode

Besides the default settings option described above, the following FTPClient instance methods control the transfer mode:

setServerPassive
    Instructs the server to enter passive mode.

setServerActive
Instructs the server to enter active mode.

copyFromFTP: *FTPServer* **file:** *remoteFileName* **toFile:** *loaclFileName* **passive:** *aBoolean*
Sets the server mode to passive (true) or active (false) for this copy operation only.

Ensuring that the connection closes

It is the responsibility of your application to ensure that the connection is closed when the FTP session ends. In the normal case, in which the transactions proceed without difficulty, and the connection remains open until the application is finished, sending a close (or disconnect) message to the FTPClient is sufficient, as shown in the examples so far.

Especially for prolonged and more complex sessions, involving multiple transactions and intermediate processing, additional steps should be taken to ensure that the connection is closed if something goes wrong.

Both normal and abnormal termination can be covered by using an *ensure block*. This involves simply enclosing the session code in a block, and following it with ensure: [ftpClient close]. For example, again slightly modifying our example code:

```
| ftpClient |

"create client, connect, and log in"
    [ ftpClient := FTPClient loginToHost: 'ftp.parcplace.com'
        asUser: 'anonymous'
        withPassword: 'bruce@parcplace.com'.
"do stuff"
    Transcript nextPutAll: ftpClient currentWorkingDirectory displayString;
        cr ;
        flush. ]
"close connection"
    ensure: [ ftpClient close ].
```

Now, whether the FTP session terminates normally, by succeeding, or abnormally by failing somewhere, the connection will be closed.

Refer to the VisualWorks *Application Developer's Guide* for more information about unwind code protection.

Handling FTP Exceptions

Wrapping your code in an ensure block is a general way of protecting against exceptions that would leave an open connection. Often, however, more specific handling of exception conditions is necessary.

For general information about exception handling in VisualWorks, refer to the *Application Developer's Guide*.

A number of exception classes are provided for trapping exception conditions, based on the FTP response. These are all subclasses of the FTPSignal class, which is itself a subclass of Exception.

Exception Class	Description	FTP Response Code
FTPConnectionSignal	There is no connection, or the connection has become stale.	none
FTPPreliminaryReply	Preliminary response. The requested action is being initiated. No further requests should be sent until a completion reply is received.	1xx
FTPIntermediateReply	Positive intermediate reply. The command has been accepted but is being held pending receipt of further information. The application should send another command specifying this information.	3xx
FTPChallenge	Positive intermediate response, but the server needs more information, usually a user name, password, or account.	3xx
FTPTransientReply	Transient negative completion reply. The command was not accepted, and the action did not take place. The condition is temporary. The application should return to the beginning of the command sequence (if any) and begin again.	4xx
FTPPermSignal	Permanent negative reply.	5xx
FTPNoSocketSignal	The request was not accepted and the action did not take place. The request sequence should not be repeated in the same order.	5xx

These exception classes cover all FTP responses except for the 2xx "success" responses.

To capture these exception classes, use the on:do: message. The receiver is a block, containing the code for which an exception is being trapped. The first argument is one of the exception classes. The second argument is a block of code to execute if the exception occurs. For example:

```
| ftpClient |
[ftpClient := FTPClient loginToHost: 'ftp.parcplace.com'
      asUser: 'anonymous'
      withPassword: 'bruce@parcplace.com'.

"do stuff"
[ ftpClient makeDirectory: 'bogus' ]
“catch exception”
      on: FTPPermSignal
      do: [ :x | Transcript show: x parameter code; cr ] . ]

"close connection"
      ensure: [ftpClient close].
```

This code raises a permanent FTP exception because the server does not grant directory creation access to anonymous accounts, and so the do: block is evaluated. The single argument to the do: block is the FTPPermSignal instance.

For most purposes, the exception classes provide sufficient information for application programming. If more detailed information is required, you can work with the FTP response itself, which is represented by an instance of FTPResponse, and is returned by the exception class in answer to the parameter message, as shown. These four messages return parts of the FTP response:

code
    Returns the FTP response code as a String.

message
    Returns the FTP response message as a String.

status
    Returns the FTP response code as a number.

statusAt: *anInteger*
    Returns the digit at the specified position (1 - 3).

The response codes returned by code, status, and statusAt: are intended for use by programs (automata), and are interpreted as specified in RFC 959. The text message is suitable for humans, so can be used in any notification you might want to display to the user.

File structure types

FTP distinguishes three types of file structures commonly used on diverse hosts: File, Record, and Page structures. VisualWorks currently implements only the File type, which is an unstructured data file.

However, VisualWorks does provide the structure: message to specify the structure. To specify the file structure, send a structure: message to the FTPClient with either $F, $R, or $P as the argument:

```
ftpClient structure: $F.
```

File data representation

When transferring files between dissimilar systems, FTP clients and servers are responsible for converting data received between the host system's data representation format and NVT-ASCII representation specified for FTP usage.

In VisualWorks these issues are simplified because VisualWorks already handles data format translation for different platforms. Consequently, even for a client application deployed on different types of systems, you do not need to provide special handling for different file data representations. VisualWorks handles that for you.

File transfer operations

Two methods are provided for transferring files between a host and a client:

retrieveFileNamed: *aString* **as:** *aFilenameOrString*
Downloads the file from the remote host to the local client.

storeFileNamed: *aFilenameOrString* **to:** *aString*
Uploads the file from the local client to the remote host.

The remote file name is specified as a String, and can be specified relative to the current directory set for the FTP session on the server.

The local file can be specified by a String, or by a Filename (or LogicalFilename or PortableFilename). Use Filename methods for setting the local current directory.

Download a file from a remote server

To download a file, send a retrieveFileNamed:as: message to the FTPClient instance. The first argument is either a Filename or a String, specifying the name of the local copy of the file. The second argument is a String specifying the remote, source file relative to the current directory on the server.

By default, file transfers are in binary mode.

```
| ftpClient |
ftpClient := FTPClient loginToHost: 'ftpserver'
    asUser: 'user' withPassword: 'password'.
ftpClient setCurrentDirectory: 'visualworks'.
ftpClient retrieveFileNamed: 'hello.txt' as: 'helloworld.txt'.
ftpClient close.
```

Upload a file to a remote server

To upload a file, send a storeFilename:as: message to the FTPClient instance. The first argument is a String specifying the name of the file as it will be stored remotely, relative to the current directory on the server. The second argument is either a Filename or a String, specifying the local file to be uploaded.

By default, file transfers are in binary mode.

```
| ftpClient |
ftpClient := FTPClient loginToHost: 'yourftpserver'
    asUser: 'zzuser' withPassword: 'zzpassword'.
ftpClient setCurrentDirectory: 'visualworks'.
ftpClient storeFileNamed: 'vwlogo.jpg' to: 'vwlogo2.jpg'.
ftpClient close.
```

Restarting a file transfer

Especially for large file transfers, it is convenient to be able to restart a transfer at the point where it was interrupted by a lost connection. NetClients provides two messages that allow you to restart a file transfer at the place where it was interrupted. These commands rely on the server supporting the FTP SIZE command. If this command is not supported, they restart at the beginning of the file.

restartRetrieveLocalFile: *localFileNameString* **fromFile:** *remoteFileNameString*
Checks the size of the (partially downloaded) local file, and begins transferring only data that has not been transferred yet.

restartStoreRemoteFile: *remoteFileNameString* **fromFile:** *localFileNameString*
Checks the size of the (partially uploaded) remote file, and begins transferring only data that has not been transferred yet.

Directory operations

Get the current directory

To get the current directory on the remote host, send a currentWorkingDirectory message to the FTPClient instance.

```
| ftpClient |
ftpClient := FTPClient loginToHost: 'servername'
      asUser: 'user' withPassword: 'password'.
ftpClient currentWorkingDirectory.
ftpClient close.
```

Create a new directory

To create a directory on the remote host, send a makeDirectory: message to the FTPClient instance. The argument is a String specifying the directory to create. The parent directory must already exist.

```
| ftpClient |
ftpClient := FTPClient loginToHost: 'ftpserver'
      asUser: 'user' withPassword: 'password'.
ftpClient makeDirectory: 'visualworks'.
ftpClient close.
```

Delete a directory

To delete a directory, send a removeDirectory: message to the FTPClient instance. The argument is a String specifying the directory to remove:

```
| ftpClient |
ftpClient := FTPClient loginToHost: 'yourftpserver'
      asUser: 'zzuser' withPassword: 'zzpassword'.
ftpClient removeDirectory: 'visualworks'.
ftpClient close.
```

The directory must exist and be empty, otherwise an FTPPermSignal is raised, e.g., “550 : The directory is not empty”.

Listing files in a directory

Two messages are provided for listing directory contents:

directoryContents: *aCollection* **do:** *aBlock*

Retrieves a Collection containing the contents of the specified directories, and passes its contents to the block for processing. The directory listing is in “long” format:

```
-rw-r--r--   1 bboyer   pps          10 Nov 26  1996 stuff.txt
```

filesInDirectory: *aCollection* **do:** *aBlock*

Retrieves a Collection containing only the file names contained in the specified directories, and passes it to the block for processing.

The directories to be listed are provided in an collection, such as an Array. To list the current directory only, give an empty collection:

```
"a single directory"
    ftpClient filesInDirectory: #( thisDirectory ) do: [ :entry | some work ].
"multiple directories"
    ftpClient filesInDirectory: #( archives newWork )
        do: [ :entry | some work ].
"current directory"
    ftpClient filesInDirectory: #( ) do: [ :entry | some work ].
```

For a single directory, it can be given as a String. The current directory, can be specified as a quoted single space:

```
"a single directory"
    ftpClient directoryContents: 'thisDirectory' do: [ some work ].
"the current directory"
    ftpClient directoryContents: ' ' do: [ some work ].
```

If a directory is specified including path name separators, it must be provided as a String:

```
"current directory and a subdirectory"
    ftpClient filesInDirectory: #( ' ' 'archives/newWork' ) do: [ some work ].
```

Pattern matching can be used to filter the list. The pattern must be specified as a String:

```
"list only files and directories starting with 's' or 't' "
    ftpClient directoryContents: #( 's*' 't*' ) do [ some work ]
```

This next example lists the current directory contents in long format and then in filename-only format, to the Transcript:

```
| ftpClient |
[ ftpClient := FTPClient loginToHost: 'ftpserver'
    asUser: 'user' withPassword: 'password'.
ftpClient directoryContents: #() do: [ :entry | Transcript show: entry; cr ].
ftpClient filesInDirectory: #() do: [ :entry | Transcript show: entry; cr ]
] ensure: [ftpClient close ].
```

File operations

Delete a remote file

To delete a file, send a deleteFile: message to the FTPClient instance. The argument is the name of the file as a String.

```
| ftpClient |
ftpClient := FTPClient loginToHost: 'ftpserver'
      asUser: 'user' withPassword: 'password'.
ftpClient setCurrentDirectory: 'visualworks'.
ftpClient deleteFile: 'hello.txt'.
ftpClient close.
```

An error will result if the file cannot be found: “550 : The system cannot find the file specified”.

Rename a remote file

To rename a file, send a rename:to: message to the FTPClient instance. The first argument is the file’s original name as a String, and the second argument is the new name as a String.

```
| ftpClient |
ftpClient := FTPClient loginToHost: 'ftpserver'
      asUser: 'user' withPassword: ‘password'.
ftpClient setCurrentDirectory: 'visualworks'.
ftpClient rename: 'hello4.txt' to: 'hellofour.txt'.
ftpClient close.
```

Get a file size

Some FTP servers will report a file size, if they support the FTP SIZE command. To get the size, send a fileSize: message to the FTPClient instance. The argument is the file name as a String.

Systems that do not support the command will raise an FTPPermSignal, which you should trap.

```
| ftpClient |

ftpClient := FTPClient loginToHost: 'ftpserver'
      asUser: 'user' withPassword: ‘password'.
[ ftpClient fileSize: 'prom.zip' ]
      on: FTPPermSignal
      do: [ :x | Transcript cr; show: x parameter message ; cr. ]

ftpClient close.
```

For systems that do not report file sizes in response to this command, you can parse the results of the directoryContents:do: command to extract the size.

Setting file transfer mode

FTP recognizes Stream, Block, and Compressed file transfer modes. NetClients currently only supports Stream mode.

If your application provides its own support for Block and Compressed modes, you can set the mode by sending a mode: message to the FTPClient instance. The argument is a character identifying the mode.

```
| ftpClient |

"S=Stream B=Block C=Compressed"
ftpClient := FTPClient loginToHost: 'ftpserver'
      asUser: 'user' withPassword: 'password'.
ftpClient mode: $S.

ftpClient close.
```

Getting server information

Display the remote server type

Some FTP servers will report information about the server, in response to the FTP SYST command. Typically, Windows servers respond, but UNIX servers do not.

To get system information where available, send a remoteSystemType message to the FTPClient instance. Trap for FTPPermSignal to catch requests to non-responding systems.

```
| ftpClient |

ftpClient := FTPClient loginToHost: 'ftpserver'
      asUser: 'user' withPassword: 'password'.
[ ftpClient remoteSystemType inspect ]
      on: FTPPermSignal
      do: [ :x | Transcript show: x parameters code ; cr ].

ftpClient close.
```

Display the remote server status

All FTP servers will report their status. The actual information returned is not specified, and is typically not very descriptive.

```
| ftpClient |

ftpClient := FTPClient loginToHost: 'ftpserver'
      asUser: 'user' withPassword: ‘password'.
(ftpClient stat) inspect.

ftpClient close.
```

Using FTP commands and responses

FTPClient also provides a more immediate interface to FTP, allowing you to issue the raw FTP commands yourself. For instance, instead of sending directoryContents:do:, you can explicitly send the FTP LIST command with the appropriate parameters.

This interface provides you more control over your FTP session, allowing options that might not be available using the higher-level interfaces already described. It also carries with it more responsibility, because you are responsible for issuing these commands, and waiting for responses, in the required sequences.

Refer to RFC 959 and auxiliary documents for command and sequence descriptions. In this section we describe the FTPClients protocol for issuing the low-level commands, and some simple examples.

Simple commands and responses

FTP commands and responses are transferred on the control port. For some FTP commands, this is the only port involved in the exchange. For these simple commands, send an executeCommand: message to an FTPClient instance. The argument is a String consisting of the FTP command, or the FTP command and its argument if one is required.

For example the following code changes the working directory and then deletes a file. Both commands take an argument. Finally, the connection is closed and disconnected. Since a permanent exception may occur if the directory or the file doesn’t exist, we trap them.

```
| ftpClient |

[ ftpClient := FTPClient loginToHost: 'ftpserver'
      asUser: 'user' withPassword: 'password'.
      [ ftpClient executeCommand: 'CWD visualworks' ]
          on: FTPPermSignal do:
              [ :x | Transcript show: x parameter message; cr ].
      [ ftpClient executeCommand: 'DELE helloX.txt' ]
          on: FTPPermSignal do:
              [ :x | Transcript show: x parameter message; cr ].
      ftpClient executeCommand: 'QUIT'

] ensure: [ ftpClient disconnect ]
```

Notice that the FTP QUIT command tells the server to close the connection. Following it with a disconnect (or quit) message to the FTPClient at this point only informs the client that the connection is closed.

Data transfer commands

Many commands, such as those that transfer files or directory listings, require a data port in addition to the control port. VisualWorks currently supports only stream data for the data port. The following messages support data transfer:

readStreamCmd: *aCommandString*
: Create a ReadStream on the data connection created by *aCommandString*. The command string begins with the FTP command, and includes any command arguments.

writeStreamCmd: *aCommandString*
: Create a WriteStream on the data connection created by *aCommandString*. The command string begins with the FTP command, and includes any command arguments.

The following opens a ReadStream to receive a directory listing and writes the results to the Transcript:

```
| ftpClient strm |

[ ftpClient := FTPClient loginToHost: 'ftpserver'
      asUser: 'user' withPassword: 'password'.
strm := ftpClient readStreamCmd: 'LIST archives'.
Transcript show: strm upToEnd ;cr.
strm close.
ftpClient executeCommand: 'QUIT'.

] ensure: [ftpClient disconnect]
```

Protecting against a disconnect

When executing a sequence of commands or a data transfer, the server may drop the control connection, resulting in a 421 reply. To simplify capturing these errors and automatically doing a retry, you can send executeSequence: to the FTPClient instance, with your low-level command wrapped in a block. This is done frequently in FTPClient methods, which you may inspect for examples.

Modifying the above example, we could (partially) protect against intermediate disconnects like this:

```
| ftpClient strm |

[ ftpClient := FTPClient loginToHost: 'ftpserver'
      asUser: 'user' withPassword: 'password'.
ftpClient executeSequence:
      [ strm := ftpClient readStreamCmd: 'LIST archives'.
      Transcript show: strm upToEnd ;cr.
      strm close.
      ftpClient executeCommand: 'QUIT'. ]

] ensure: [ftpClient disconnect]
```

3

Internet Messages and MIME Types

Introduction

Internet messaging and electronic mail standards were defined initially to consist exclusively of plain, US-ASCII text. Such text messaging standards are defined in RFC 822, and is the basis for almost all subsequent internet mail and messaging standards.

A text message, as defined in RFC 822, consists of two parts: a header and a body. The header is a collection of fields (date, from, subject). The body, which is optional, consists simply of lines of text.

This representation is too limiting for many purposes, however, so internet messaging standards have been extended beyond the capabilities of plain text. MIME types, defined in RFCs 2045-2048, provide a standard for specifying other kinds of message body content, such as non-US-ASCII text, binary data, such as executables and images, and multi-part messages. In short, MIME

- defines the structure of multi-part messages
- provides a mechanism for specifying the type of information in the message (content-type)
- describes how non-ASCII information can be transported
- describes how non-ASCII information can be encoded in message headers
- defines 7-bit transport and content transfer encoding

This chapter describes the support provided by VisualWorks for handling internet messages and MIME types. These capabilities form the foundations for the messaging services support described in subsequent chapters.

MIME support classes

NetClients provides an extensive framework in the MIME parcel. Of those many classes, the following are essential for working with MIME:

```
MessageElement
    MimeEntity
        RFC822Message
            MailMessage
        MimeMessageBody
            MultipartBody
            SimpleBody
    HeaderField
        StructuredHeaderField
            CollectionField
                CacheControl
        ScalarField
            DateField
            MailboxListField
            NumericField
            SingleMailboxField
            VersionField
        ValueWithParametersField
            AcceptHeaderFields
                AcceptCharsetField
                AcceptField
                AcceptLanguageField
                ManOptField
            AuthenticateChallengeField
            SingleValueWithParametersField
                ContentDispositionField
                ContentTypeField
```

RFC822Message and MailMessage are the enclosing message entities to which other MIME entities are added. MimeEntity provides all the basic support for defining MIME entities. RFC822Message adds specific protocol for a large number of header fields to a text message. MailMessage further adds protocol specifically for dealing with complex message bodies, including attachments.

These classes provide the protocol you need to use. The remaining classes encapsulate the syntax and semantics of MIME messages, doing the following work:

- Parses the message according to various protocols (RFC822, MIME, HTTP)
- Decodes encoded headers

- Provides messages for setting and getting most common header fields, including those with parameters

Protocols that send MIME messages build on the basic MIME protocol, adding further specialized functionality. For example, in all the sample code found in the chapters pertaining to e-mail, you will find reference to the MailMessage class. It is used for both sending and receiving electronic messages and is, indeed, the "backbone" within VisualWorks that enables Internet messaging.

Creating Mime Entities

A MIME entity is either a header element or a body element in some enclosing message, either a mail message or an HTTP message. Accordingly, they are created for inclusion in some top-level container, the message.

For example, you might start with a basic RFC 822 style message:

```
msg := RFC822Message new
```

This creates a very basic message. You can inspect the result to examine its structure. To this basic structure you add MIME entities as needed.

MailMessage provides support for additional message entities that are typically used only by mail messages, such as attachments.

Of particular interest for this chapter is that it has a header instance variable, which holds a set of header entities (initially empty), and a value instance variable, which holds its body, if any.

A better way to create a new MIME entity is to send one of these instance creation messages to MimeEntity or a subclass:

source: *aStream*
Creates a simple MIME entity from the contents of the Stream.

fileName: *aStringOrFilename* **withEncoding:** *aSymbol*
Creates a simple MIME entity containing the contents of the file, guessing at the MIME type from the *afileName* extension. If *afileName* does not have an extension, returns default 'application/octet-stream'.

newTextHTML
Creates a new entity with type text/html.

newTextPlain
Creates a new entity with type text/plain.

readFrom: *aStream*
Creates a new entity from *aStream*, parsing the information, such as message headers, based on its structure.

readHeaderFrom: *aStream*
Creates a new entity containing the *aStream*, parsing the header information only.

The result is a new MimeEntity with the appropriate MIME type specified in the entity header. These will be used in later examples.

Adding header fields

MIME defines a few specific header fields and permits others. Net Clients supports adding any of these to the header.

Headers that are defined by MIME are supported by specific set/get accessors. A sampling of the setters is (browse the MimeEntity accessing method category for more, and for getters):

charset: *aString*
Set the 'charset=*aString*' to the Content-type header.

contentId: *aString*
Add or set the 'ContentId:' header to *aString*.

contentLength: *anInteger*
Add or set the 'Content-length:' header to *anInteger*.

contentTransferEncoding: *encodingNameString*
Add or set the 'Content-transfer-encoding:' header to *encodingNameString*.

contentType: *aString*
Add or set the 'Content-type:' header to *aString,* with the default charset.

mimeVersion: *aString*
Add or set the 'Mime-version:' header to *aString*.

addMimeVersion
Add or set the 'Mime-version:' header with the default version string.

Any of these can be sent to the message already created to set the header. For example:

```
| message |
message := RFC822Message new.
message addMimeVersion.
message contentType: 'text/xml'.
```

The messages create the appropriate instances of HeaderField subclasses and add them to the header.

Some header fields take optional parameters and other information. These are implemented as subclasses of ValueWithParametersField, and provide additional messages for supplying the additional information. For example, ContentTypeField represents the **content-type** field in MIME and HTTP protocols, which may specify a type, a subtype, may specify a character set for text types, and boundary for multipart subtypes, and so on. If you use these header fields, browse the class for supporting messages.

Additional fields can be added to the header for informational purposes. This is done by creating a HeaderField with a value and adding that to the message header. For example:

```
| message header |
message := RFC822Message new.
header := HeaderField name: 'MyField'.
header value: 'dummy'.
message addField: header.
```

Another useful message is getFieldAt:, which takes a field name as a String for its argument. If the field already exists, the message returns the field and its value. If the field does not yet exist, it is created, and you can then provide a value for it. The following expression checks for a field's existence, creates it if necessary, and sets its value:

```
| message |
message := RFC822Message new.
(message getFieldAt: 'bogus' ) value isNil
    ifTrue: [ message fieldValueAt: 'bogus' put: 'stuff' ]
```

Net Clients email and HTTP support provide protocol for adding headers that are meaningful for specific types of messages, as described in later chapters.

Adding a body

Instances of MimeEntity and its subclasses include a value instance variable that contains the message body, which holds either a SimpleBody or a MultipartBody (which are subclasses of MimeMessageBody).

Simple Body

Initially value holds an empty SimpleBody. To access the body, send a body message to the message:

```
message body
```

To give the body a simple content, such as a String, send a contents: message to it:

```
message contents: 'some text'
```

The value could also be read from a file:

```
message contents: 'test.txt' asFilename
```

A better way to specify the body is when the message is created initially, by using the source: instance creation method:

```
message := MimeEntity source: ‘some text’ readStream.
```

There are a couple other instance creation methods, defined in MimeEntity (such as readFrom: illustrated below), that do the same. This has the advantage of creating the message with the correct content type, and creating the SimpleBody with value ‘some text’.

The following is a more complete example of creating a MIME mail message using this last approach:

```
| message messageString |
    messageString := 'Date: 27 Aug 76 09:32 PDT
From: Jon Doe <JDoe@This-Host.This-net>
Subject: Re: The Syntax in the RFC
Sender: KSecy@Other-Host
Reply-To: Sam.Irving@Reg.Organization
To: George Jones <Group@Some-Reg.An-Org>,
      Al.Neuman@MAD.Publisher
Message-ID: <4231.629.XYzi-What@Other-Host>

A bunch of text.
'.

message := MimeEntity readFrom: messageString readStream.
```

This specifies the entire message, including message headers (Date, From, Subject, etc.) as a String in the form specified for a MIME message. The last line uses the readFrom: instance creation method to create the MIME message from the String.

Multipart Body

MIME provides the capacity of multipart messages, consisting of multiple MIME body elements preceded, followed, and separated by a boundary marker. NetClients represents a multipart message as a message with a MultipartBody, which holds a collection of MimeEnties as body parts.

To create a multipart body, simply send an addPart: message to the MIME message with a MimeEntity as the argument:

```
| message newEntity |
message := MailMessage source: 'some text' readStream.
newEntity := MimeEntity source: 'some other text' readStream.
message addPart: newEntity.
```

If the MIME message currently has a SimpleBody, addPart: mutates the current body to a MultipartBody, with the original body as the first body part, and adds the new element to the collection of body parts. If the body is already a MultipartBody, the new part is simply added to the collection.

As explained above, using the source:, or other MimeEntity instance creation methods, invokes the MIME framework to provide the correct content type.

Body parts are separated by a “boundary.” A default boundary is provided, but you probably want to specify your own. After the body has been converted to multipart, you can specify the boundary by sending a boundary: message to the MIME message:

```
msg boundary: 'some_string_that_will_not_occur_naturally’
```

The boundary delimiter begins with -- (two hyphens), which are provided by VisualWorks. The String may be up to 70 characters, and must be such that it does not occur in any of the embedded elements. Any strategy to create such a String can be used.

Creating a File Attachment

You can create a file attachment for an external file by sending the fileName:withEncoding: instance creation message to MimeEntity. For example:

```
attachment := MimeEntity fileName: 'visual.im.zip'
    withEncoding: #'Windows-1252'.
```

The entity is created with the proper content type. You can then add the entity to a message as a part, as described above.

Sending a Message

Normally, a message is constructed and then sent by an appropriate client (HTTP, SMTP, etc.) using the message protocol for that client (e.g., sendMessage: for SMTPClient). Refer to the chapters for individual protocols later in this document.

As a convenience, MailMessage defines a send message, which sends the message via the default SMTP server, if one is defined. The server is defined in the Net Settings tool. A default user may also be specified, but is not required if the message From: line has a legal mail address (name and domain).

For example, if a default SMTP server is specified in Net Settings, this should succeed:

```
| message |
message := MailMessage newTextPlain.
message from: 'santa@northpole.net';
      to: 'jdoe@abc.com';
          subject: 'Start making your list now';
          text: 'What would you like for Christmas?';
          yourself.
message send.
```

If a default user is also specified, then the from: 'santa@northpole.net' can also be omitted.

Reading a MIME Message

Reading a MIME message entails accessing headers and body parts as needed.

Accessing headers

Several messages are provided for accessing header fields.

The following messages, and variations of them, are defined in the accessing field and body parts message category in MimeEntity. These provide general access to fields in MIME entities.

fieldValueAt: *aString*
: Returns the value of the header field named *aString*, or nil if it doesn’t exist or is empty.

fieldsAt: *aString*
: Returns a List of all header fields named *aString*. This is useful if there are multiple header fields with the same name, as is permitted by MIME.

getFieldAt: *aString*
: Returns the value of header field *aString* if it exists; otherwise creates the field.

For example, given the complete MIME message example above, the following will return the subject line as a String:

```
mimeMsg fieldValueAt: 'subject'
```

Notice that the header fields have initial lower-case names.

There are specific messages for accessing particular fields as well, especially content related fields. These are implemented using the above methods, providing an easier to use interface. Browse the

contentId
: Returns the value of the **`content-id`** field.

contentLength
: Returns the value of the **`content-length`** field.

contentType
: Returns the value of the **`content-type`** field.

RFC822Message defines additional messages for accessing headers of specific interest in mail messages. Browse the **accessing** method category for the complete set. The following is a small sample of the available messages.

to
: Returns the contents of the **`to`** field.

from
: Returns the contents of the **`from`** field.

bcc
: Returns the contents of the **`bcc`** field.

cc
: Returns the contents of the **`cc`** field.

date
: Returns the value of the **`date`** field.

replyTo
: Returns the value of the **`reply-to`** field.

For example, given a mail message, you request any of this header information from it:

```
| message mimeMsg addressee |
message := popClient retrieveMessage: 1.
mimeMsg := MailMessage readFrom: message readStream.
^mimeMsg to.
```

Accessing bodies

A MIME message may have a single-part (SimpleBody) or multipart (MultipartBody) body, or no body at all.

The basic MimeEntity protocol for accessing bodies consists of these messages:

contents
For a simple message, returns a ByteString containing the message body. For a multi-part message, returns an OrderedCollection of the parts, which are instances of MimeEntity.

parts
Returns an OrderedCollection of the message parts of a multi-part message (same as contents). The collection is empty for a simple message.

partAt: *partIndex*
Returns the MimeEntity at the specified *partIndex* of the collection returned by parts.

To test whether the body is simple or multi-part, send an isMultipart message to the message. The response is a Boolean which you can use to choose how to process the message. For example, this returns the message text as a ByteString for either:

```
message isMultipart ifTrue:
    [ ^(message partAt: 1) contents ].
    ^message contents.
```

Given a multi-part message, which will be an instance of MailMessage, there are additional messages you can send it to operate on its text part or parts. Browse the **accessing** and **utility** method categories for these operations.

For example, texts are often included in a variety of formats, such as **`text/plain`** or **`text/html`**, providing the mail reader a choice of text formats to use for display purposes. When there are alternatives, the text type itself is **`multipart/alternative`**, and each format is provided as a part.

The following messages allow you access these formats and parts.

allAlternativeTextFormats
Returns a collection of all alternative text formats.

allTextParts
Returns a collection of all text parts.

prepareForTransport
Verifies that fields are in proper order, that required fields are present, that assigned boundaries are present, and that attachments are base64 encoded.

replaceTextWith: *aString*
Replaces the message body with *aString*, and resets the content type.

saveAttachment: *aMimeEntity* **on:** *aStream*
Writes *aMimeEntity* on *aStream* using appropriate encoding.

saveAttachmentAt: *anIndex* **on:** *aStream*
Writes the message attachment at *anIndex* on *aStream* with appropriate encoding.

saveTextOn: *aStream*
Writes the message text on *aStream*. If the text includes alternates, all are written in the order they occur.

saveTextOn: *aString* **inPreferenceOrder:** *anArrary*
Writes all alternate message texts on *aStream* in the order specified by *anArray*, e.g., #('html' 'plain' '*')

text
Returns the plain text part of a message text.

textInPreferenceOrder:
Returns text in all alternative formats, in the specified order.

Get content information of a message part

Message parts are MIME entities, and so you can get the usual MIME information from it. For example, the following code checks a multi-part message to verify that the first part is plain text, and if so, returns the text:

```
| msg message plainText |
msg := mailClient retrieveMessage: 1.
message := MailMessage readFrom: msg readStream.
message isMultipart ifTrue:
    [ ( ( message partAt: 1 ) contentType = 'text/plain' ) ifTrue:
        [ ^plainText := ( ( message partAt: 1 ) contents ) ] ].
```

Extract the best text representation

Some clients can send a message in alternative formats as separate parts, such as plain text and HTML. The message itself has content type **`multipart/alternative`**. Alternates are, by convention, placed in the message in order of increasing complexity, so plain text comes first.

Clients that can handle multiple formats, such as plain test and HTML, can specify their own preference order for the version to display, by sending textInPreferenceOrder: to the mail message with an ordered collection of preferences. So, if your client displays HTML, you may prefer that format over plain text, but still use plain text if there is no HTML alternate. Follow the list with asterisk (*) to accept any format in the absence of a preferred format. This example extracts html if available, plain text if not, and the first available if neither of those is available.

```
| msg message |
msg := mailClient retrieveMessage: 1.
message := MailMessage readFrom: msg readStream.
^message textInPreferenceOrder: #('html' 'plain' '*').
```

Message transfer encoding

MIME entities can be encoded using the encoding scheme specified in the content-transfer-encoding field. Encoding is necessary to encode data for transfer over some transfer protocols, such as SNMP which restricts mail message data to 7-bit US-ASCII.

Net Clients supports the two standard encoding schemes, base64 and quoted-printable, which transform messages to conform to the 7-bit US-ASCII restriction. Other encodings can be specified, but Net Clients does not provide the encoding.

Quoted-printable is used for message bodies that are mostly 7-bit US-ASCII, but may contain a few 8-bit characters. Any 8-bit characters are transformed to be represented as “=” followed by the two-digit hexadecimal representation.

Base64 encoding is used for message bodies if quoted-printable is not appropriate. It represents arbitrary octets in a 65 character subset of US-ASCII. Base64 encoding can be used, for example, to include binary data in a mail message, such as in an attachment.

By default when a MIME message is written, base64 transfer encoding is applied. So,

```
MailMessage new
    addFileAttachment: 'visual.zip';
    subject: 'Sending zip file';
    send.
```

will send the message attachment base64 encoded.

The default encoding option can be set to false by sending:

```
MimeWriteHandler applyTransferEncoding: false
```

Net Clients protocol for encoding and decoding is very simple. These messages are sent to a MimeEntity, typically a SimpleBody:

addContentTranferEncoding: *anEncoding*
Modifies the receiver MimeEntity to include the **content-transfer-encoding** header for *anEncoding*, and encodes the body as appropriate. The String specifying *anEncoding* is not validated, but must be either 'base64' or 'quoted-printable' (case sensitive) for the body to be encoded. If you do not add the content transfer encoding for mail message attachments that are not plain text, the mail client adds default encoding 'base64' when the message is sent.

removeContentTransferEncoding
Removes the **content-transfer-encoding** header and decodes the body if it had been either base64 or quoted-printable encoded.

For example, if you added a part that should be encoded (and did not use a message like addFileAttachment: that handles the encoding), you can access the part and apply the encoding:

```
( mailMessage parts at: 2 ) addContentTransferEncoding: ‘base64’
```

Removing content transfer encoding is a configurable setting.

```
Settings mailRemoveContentTransferEncoding: true.
```

The default setting is false. If set to true, content transfer encoding is removed while parsing the message.

To remove the encoding from a part of a received message, send:

```
( mailMessage parts at: 2 ) removeContentTransferEncoding
```

International characters in header fileds

To send messages with header fields that have non-ASCII characters, the field values should be represented as encoded-word's:

encoded-word = "=?" charset "?" encoding "?" encoded-text "?="

The charset portion specifies the character set associated with the unencoded text. A charset can be any of the character set names allowed in an MIME "charset" parameter of a "text/plain" body part, or any character set name registered with IANA for use with the MIME text/plain content-type.

VisualWorks supports encoding values "Q" and "B". The "B" encoding is identical to the "BASE64" encoding defined by RFC 2045. The "Q" encoding is similar to the "Quoted-Printable" content transfer encoding defined in RFC 2045.

To correctly encode national characters in the header fields, set the header character set by sending headerCharset: to the mime entity.

```
MailMessage new
   date: Timestamp now;
   headerCharset: 'iso-8859-2';
   from: '"Žluva Tůma Řízek" <xxx@server.cz>';
   to: '"Božidar Šlapetko" <yyy@server.de>';
   subject: 'Příliš žluťoučký kůň úpěl ďábelské
  ódy.';
   charset: 'iso-8859-2';
   text: 'PŘÍLIŠ ŽLUŤOUČKÝ KŮŇ ÚPĚL ĎÁBELSKÉ ÓDY.';
   yourself.
```

If the header character set is not provided and the header field contains non-ASCII characters, the mime entity makes a guess about the charset, based on the string contents. (See MimeEncodedWordCoDec class method findEncodingFor:.)

Processing non-ASCII data

The NonASCIICharacter exception is raised while parsing a message or header with non-ASCII characters.

To simply accept non-ASCII characters without raising an exception, send an acceptNonAsciiCharacters: message to the MimeParserHandler, or appropriate subclass. By default NonASCIICharacter will be raised.

For example, this message contains the copyright character, which is not a valid in ASCII character. By default the NonASCIICharacter will be raised:

```
bytes :=
'Subject: Cincom©
Content-Type: text/plain

1234 ' asByteArrayEncoding: 'ISO-8859-1'.
stream := EncodedStream on:
    bytes readStream encodedBy: (StreamEncoder new: #'ISO-8859-1').
[ MailBuildHandler readFrom: stream
    ] on: NonASCIICharacter
        do: [ :ex | ex parameter ].
```

To make the parsing process proceed accepting the character as is, set the acceptNonAsciiCharacters option to true:

```
stream := EncodedStream on: bytes
    readStream encodedBy: (StreamEncoder new: #'ISO-8859-1').
message := MailBuildHandler new
    acceptNonAsciiCharacters: true;
    readFrom: stream.
```

To parse individual header fields with non-ASCII characters, read the header using a readFrom:acceptNonAsciiCharacters: message:

```
string := 'Received: (from Cincom©)  Tue, 18 Apr 89 23:29:47 +0900'.
HeaderField readFrom: string readStream acceptNonAsciiCharacters: true.
```

The same default behavior, raising the NonASCIICharacter exception, applies to parsing individual fields as well:

```
[ HeaderField readFrom: string readStream
    ] on: NonASCIICharacter do: [ :ex | ex parameter ].
```

When constructing a new mail message, header field values with non-ASCII characters are accepted. They are properly encoded using the specified encoding (ISO-8859-1 by default) and processed according to the MIME standard when the message is written out.

Note that if the default encoding cannot handle provided characters, a different encoding should be explicitly specified using the headerCharset: message. In the following example, the ISO-8859-2 encoding is used to encode the accented Czech characters.

```
message := MailMessage new
   headerCharset: 'iso-8859-2';
   from: '"Žluva Tůma Řízek" <xxx@server.cz>';
   to: '"Božidar Šlapetko" <yyy@server.de>';
   subject: 'Ahoj';
   text: 'text';
   yourself.
stream := String new writeStream.
message writeOn: stream.
stream contents
```

In general you can parse MIME messages from any kind of stream (internal or external). However, to facilitate parsing MIME messages straight from external streams, the parser works from bytes (not characters) at the lowest level. This assumption causes certain complications when working with internal streams on Strings as the underlying collections. (Internal streams on top of ByteArrays do not have any issues and behave exactly the same as external streams). These “character streams” are automatically wrapped in a DecodedStream which converts characters to bytes using ISO-8859-1, the character set prescribed by the MIME standard. Because of this it is possible for the DecodedStream to encounter a character that it cannot encode. Although such message is technically invalid (such characters should be encoded differently using valid characters) it is still possible to finish parsing the message. The stream is set up with a special stream error policy, the ReplaceUnsupportedCharacters policy, which signals an UnsupportedCharacterReplacement notification for these characters and by default replaces them with ASCII character NUL (code 0). This allows the MIME parsers to recover and continue. This notification can also be trapped by an application level handler and resumed with a different replacement character if so desired. The example below replaces the (invalid) trademark character with an underscore:

```
string :=
'Content-Disposition: attachment; filename="Cincom™ .txt"
Content-Type: text/plain; name="Cincom™ .txt"

some bytes
'.
   [ MimeBuildHandler readFrom: string readStream.
      ] on: UnsupportedCharacterReplacement
         do: [ :ex | ex resume: $_  ].
```

Several web browsers send Http message with non-ascii characters in parameter values. For example the filename parameter in Content-Disposition field can be sent in utf-8 encoding only. The parser can be configured to try to decode these values. To set this option, send, for example:

```
ValueWithParams defaultParameterValueEncoding: ‘utf8’
```

specifying the decoder. By default the value is nil, indicating that such characters will not be decoded.

Unknown Encoding

The `UnknownEncoding` exception is raised when the message content-type specifies an unknown encoding, either because such encoding does not exist, or because a corresponding encoder is not available in the image at that time.

This exception can be resumed, either with an explicitly specified encoding (e.g., `resumeWith: #'US-ASCII'`) or without a parameter (`resume`). In the latter case the default encoding is applied (ISO-8859-1). Here is an example (note the charset value in the Content-Type header field):

```
input :=
'From: Fred Foobar <foobar@Blurdybloop.COM>
Subject: afternoon meeting
To: mooch@owatagu.siam.edu
Content-Type: text/plain; charset=klingon

 Hello Joe, do you think we can meet at 3:30 tomorrow?
'.

"Default handling: if the exception is simply resumed, the message body
source will use ISO8859-1"
[ message := MailFileReader readFrom: input readStream.
      ] on: UnknownEncoding
          do: [ :ex | ex resume  ].
      message contents

"Overriding the default encoding: to force the message body source to use
US-ASCII encoding"
[ message := MailFileReader readFrom: input readStream
      ] on: UnknownEncoding
          do: [ :ex | ex resume: #'US-ASCII' ].
```

4

HTTP

Introduction

HTTP (Hypertext Transfer Protocol) is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a request/response protocol, in which a client sends a request to the server, and the server responds with a status line followed by a MIME-like message, most frequently a web page.

HTTP within VisualWorks mimics Filename. You can access web pages (URLs) just like local files. This provides a level of transparency in accessing HTTP-based documents. For example, you could access a financial web site and retrieve the latest stock quote for a particular ticker by parsing the stream of HTML that is returned.

HTTP support level

VisualWorks provides sufficient features to implement an unconditionally HTTP/1.1 compliant client. However, VisualWorks does not enforce HTTP/1.1 compliance of the client application.

In this chapter we explain the features available in VisualWorks and their use, with pointers to developing a client that is conditionally HTTP/1.1 compliant (implementing the MUST, but not the SHOULD, requirements of RFC 2616). However, we do not attempt to cover every requirement to make your application unconditionally compliant. Refer to RFC 2616 for the complete requirements.

Default settings

Default settings for HTTP are configurable in the Settings tool or using the following class methods.

In class Net.Settings

httpKeepAlive: *aBoolean*
If true, keeps the connection alive between requests; closes after each request otherwise.

httpRedirectRequest: *aBoolean*
If true, allows HttpMoveError (301 and 302) responses to forward to the relocated site; forwarding is blocked otherwise.

httpProxyHost: *aHostSpec*
Sets the HTTP proxy to *aHostSpec*.

httpProxyExceptions: *aList*
Sets a list of server names, as an ordered collection of strings, which the client does not use the proxy to access.

httpUser: *aNetUser*
Sets the default user to *aNetUser*, an instance of NetUser.

In class Net.HttpProtocolInterpreter

enableCookieProcessing: *aBoolean*
If true, enables cookie processing; disabled otherwise.

In class Net.CookieAgent

cacheCookies: *aBoolean*
If true, allows caching cookies; disallowed otherwise.

useCachedCookies: *aBoolean*
If true, allows using cached cookies; disallowed otherwise.

enableLimits: *aBoolean*
If true, checks limitations on cookie entries and size.

numberEntries: *aNumber*
Sets a limit on the number of entries in a cookie.

numberEntriesPerServer: *aNumber*
Sets a per-server limit on the number of entries in a cookie.

numberBytesPerCookie: *aNumber*
Sets a byte size limit on cookies.

These default settings can all be overridden for a client instance.

Implementation Classes

HTTP client support is implemented in a large number classes. This chapter focuses on those few classes that are necessary for building a client. The following is a brief overview of the most central classes. Other classes, though not all, are mentioned throughout the chapter as appropriate.

HttpClient class

HttpClient is the main class for modeling a client session. A HttpClient instance establishes and maintains a connection with the server. It holds a request, and is responsible for issuing that request and receiving back the response. It also handles some “unsuccessful” response conditions, such as a challenge for server or proxy authorization.

HttpRequest class

HttpRequest models a request. Accordingly, it holds the HTTP request line, header, and body. It supports a flexible set of messages for constructing a request in preparation for the HttpClient to send.

HttpResponse class

HttpResponse models a request response, containing a status line, headers, and a simple or multi-part body. It’s messages give easy access to each of these elements.

HttpException class

HttpException is the superclass for all HTTP exception conditions. Subclasses represent more specific conditions. Trapping the appropriate condition is the core of HTTP error handling.

HttpEntity class

HttpEntiry is the superclass for HttpRequest and HttpResponse. It implements the shared behavior.

Secure HTTP

The HTTPS parcel adds extentions that integrate SSL functionality with HttpClient, providing the ability to conduct an HTTP conversation over a secure channel established using SSL. A URI with the “https://” prefix invokes this functionality without the need for a separate client class.

Note that secure usage of SSL requires certain level of understanding of the issues involved. Therefore, we urge you to consult the available SSL documentation before relying on security of HTTPS. Refer to the *Security Guide* for more information on SSL support in VisualWorks.

Connecting to an HTTP server

Creating a connection

For most uses, the connection information is gathered from the HttpRequest (see Requesting an HTTP document), and the connection is made when the request is executed (see Sending the request).

However, it may on occasion be desirable to control a connection more explicitly. To create a connection to an HTTP server, send one of these messages to HttpClient:

connectToHost: *aString*
Establishes a connection to the host named *aString* on port 80.

connectToHost: *aString* **port:** *anInteger*
Establishes a connection to the host named *aString* on port *anInteger*.

host: *aString* **port:** *anInteger*
Identifies the host name as *aString* and the port number as *anInteger*, but does not establish a connection.

connect
Establishes a connection to the current host and port, and returns a Stream on the connection.

Connection persistence

HTTP 1.0 connections were maintained for only a single transaction, retrieving a single URI. As HTTP transactions have become more complex, for example, requesting in-line images and other associated data, this strategy has proven to be inefficient.

In HTTP/1.1 persistent connections are the default. The server may (but need not) assume that the connection persists, unless the client includes a **`Connection`** header with a connection-token **`close`**.

The persistence state of an HttpClient is stored as a Boolean in the HttpClient keepAlive instance variable. By default it is set to false in VisualWorks for a new client, so you may need to set it to true for a HTTP/1.1 client. When the request is sent, the appropriate connection-token is added to the request, indicating whether the connection is to close or remain alive, based on the keepAlive value.

Use the following protocol to set and retrieve the value:

keepAlive: *aBoolean*

Set the connection persistence state to aBoolean. Set to true for persistent, false for non-persistent.

keepAlive

Return a Boolean indicating the connection persistence state.

If a persistent connection is used, the application is responsible for closing the connection when it is finished retrieving data.

Closing a connection

To close the connection, send a close message to the client:

```
client := HttpClient new.
" do some work ".
client close
```

Clients and servers can indicate that a connection is being closed by including a **`Connection`** header field with a **`close`** token. A client must not send any more requests on a connection once it has indicated that the connection is closed.

Requesting an HTTP document

VisualWorks represents an HTTP request by an instance of HttpRequest. A request consists of:

- An HttpRequestLine instance (representing the Request-Line), consisting of:
  - an HTTP request method (e.g., GET or POST)
  - a request-URI (e.g., **`http://www.somecorp.com/index.html)`**
  - a protocol (e.g., HTTP/1.1)
- A collection of headers
- A message body (if any)

Creating a basic request

There is a single, general HttpRequest instance creation method:

method: *aMethodString* **url:** *requestURIString*
Returns an HttpRequest instance with an HttpRequestLine for the HTTP method *aMethodString*, and default version HTTP/1.1. *requestURIString* must include the protocol, host, and absolute path of the resource. The absolute path is included in the HttpRequestLine, and the host is held in a HttpRequest Host: header field.

For example:

```
req := HttpRequest method: 'GET' url: 'http://www.cincom.com'
```

For the main HTTP commands, there are special instance creation methods:

delete: *urlString*
Returns a DELETE request.

get: *urlString*
Returns a GET request.

headers: *urlString*
Returns a HEADERS request.

post: *urlString*
Returns a POST request.

put: *urlString*
Returns a PUT request.

Using this method, the following is equivalent to the previous example:

```
req := HttpRequest get: 'http://www.cincom.com'
```

Browse the Http commands class method category in HttpRequest for additional methods.

Modifying a request

Once you have an HttpRequest instance, you can modify or add parts to it as necessary.

Change the version number

The basic request created above includes an HttpRequestLine and a (initially empty) collection of headers. The request line holds the method name, the absolute path (without the host), and the protocol version. So, for the example above, the request line is:

```
GET / HTTP/1.1
```

Since no path was given in the creation command, but only the host name, the absolute path is simply /.

You may need to change the HTTP version level, which you can do with:

version: *aVersionString*
Set the protocol version to *aVersionString.*

So, if you are using HTTP/1.0, change the default version by sending:

```
req version: 'HTTP/1.0'
```

Add header fields

There are a large number of header fields that can be inserted into an HTTP request. Most of these can be added using messages provided by HttpRequest and its superclasses. Browse the accessing fields and similarly named message categories.

For example, the **`Accept`** request-header is used to specify a range of acceptable media types. To specify types, send the accept: message to the HttpRequest with either a String or an AcceptField instance, specifying the types:

```
req accept: ‘audio/basic’
req accept: (AcceptField media: 'audio/basic')
```

When specified as a String creates an AcceptField header, if it doesn’t already exist in the request, and adds the String to the list of accepted types. If the AcceptField already exists, the String is simply added to the list. However, when specified as an AcceptField, the message replaces the entire AcceptField with the new one.

The valid values for the header fields are specified in RFC 2616. VisualWorks automatically adds some header fields as part of other operations, such as adding a message body.

Add a simple message body

Some requests, such as PUT messages, may have a message body, which may be encoded. Requests that include a body must indicate this by including either a **`Content-Length`** or **`Transfer-Encoding`** header field.

To add a body, send a contents: message to the request. Using this approach, you are responsible to supply any necessary header. The content length is added later by the message writer.

The argument can be a String, in the case of a text body, for example:

```
req contentType: ‘text/plain’
bodyText := 'This is a test body'.
req contents: bodyText.
```

To provide a body from a file, whether the file is text or binary, you can provide a Filename. More extensive header information is generally required:

```
req contentType: 'application/octet-stream;name=visual.im;
    charset=iso-8859-1'
req contents: 'visual.im' asFilename
```

It is easier, however, to include binary parts in a multipart body using addPart:, which provides the appropriate header information.

Adding a multi-part body

Creating a multipart body uses the mechanism provided by MIME support, as described in Chapter 3, “Internet Messages and MIME Types.”

To summarize, you create a new MimeEntity by sending, for example, a fileName: instance creation message to MimeEntity. The result is a new MimeEntity with the appropriate MIME type specified in the entity header. You then add the part to the request body as a new message part:

```
req := HttpRequest put: 'http://some.server/'.
req contents: 'some text'.
newPart := MimeEntity fileName: 'c:\XMLSources\myXMLfile.xml'.
req addPart: newPart.
```

In this case, the request already had a SimpleBody, and the addPart: message converts the request body to a MultipartBody, adjusts the request header, makes the original body a part, and adds the new part. It also declares the required boundary delimiter in the request header and inserts it where required, before each part and after the last part.

Sending the request

Once you have built the request, you instruct the HTTP client to send the request to the server by sending it an executeRequest: message, with the request as the argument. For example, the following retrieves the Cincom home web page:

```
| httpClient request response |
httpClient := HttpClient new.
request := HttpRequest get: 'http://www.cincom.com'.
response := httpClient executeRequest: request
```

The returned value of the executeRequest: message is an HttpResponse, which we hold for processing.

This simple request uses the default settings for message chunking (true) and compression (false). Refer to “Chunking Large Attachments” on page 4-18 and “Compression” on page 4-19 for information about controlling these featuers.

Posting Form Data

The capabilities for submission of HTML form data support both simple "url encoded" format in a single-part HTTP request (content-type: application/x-www-form-urlencoded), and data submission as an individual parts in a multi-part HTTP request (content-type: multipart/form-data). These options are provided as extensions to HttpClient and HttpRequest, loaded from the WebSupport package.

Multi-part messages are used when form data contains entries with relatively large values, for example when a form has external files attached to it for upload to the server.

The default behavior is to submit forms as simple requests. Form entries can be added individually using #addFormKey:value: message, or set at once using #formData: message, which takes a collection of Associations. Note that #formData: replaces any previous form content.

```
stream := String new writeStream.
(HttpRequest post: 'http://localhost/xx/ValueOfFoo')
    addFormKey: 'foo' value: 'bar';
    addFormKey: 'file'  value: 'myFile';
    writeOn: stream.
stream contents
```

An alternative way to post a form is using HttpClient. In this case the request is immediately executed and the result is returned from the server.

```
HttpClient new
    post: 'http://localhost/xx/ValueOfFoo'
    formData: (
        Array
            with: 'foo' -> 'bar';
            with:'file' -> 'myFile').
```

To force the form to submit as a multipart message, send #beMultipart to the request at any point. Any previously added entries are automatically converted to message parts. Note however that conversion of multipart messages back to simple messages is not supported, as it is not always possible without a potential loss of information.

```
stream := String new writeStream.
(HttpRequest post: 'http://localhost/xx/ValueOfFoo')
    addFormKey: 'foo' value: 'bar';
    beMultipart;
    addFormKey: 'file'  value: 'myFile';
    writeOn: stream.
stream contents
```

File entries can be added using message addFormKey:filename:source:. Adding a file entry automatically forces the message to become multipart, to be able to capture both the entry key and the filename.

```
stream := String new writeStream.
(HttpRequest post: 'http://localhost/xx/ValueOfFoo')
    addFormKey: 'foo' value: 'bar';
    addFormKey: 'text'  filename: 'text.txt'
        source: 'some text' readStream;
    writeOn: stream.
stream contents
```

Adding a file entry attempts to guess the appropriate Content-Type for that part from the filename extension. If it does not succeed, the content type is set to default, application/octet-stream. File names with non-ASCII characters are automatically encoded using UTF8 encoding. UTF8 is also used for the file contents if the source is a character stream (as opposed to byte stream).

Adding an entry to a multipart message returns the newly created part. This allows you to modify any of the default settings or to add new ones. For example, this changes the filename and file contents encoding to ISO8859-2:

```
stream := String new writeStream.
request := HttpRequest post: 'http://localhost/xx/ValueOfFoo'.
part := request addFormKey: 'czech'
    filename: 'kú .txt'
    source: 'P íli   lu ou ký kú  úp l  ábelské ódy.' withCRs readStream.
part headerCharset: #'iso-8859-2';
    charset: #'iso-8859-2'.
request writeOn: stream.
stream contents
```

To parse messages containing forms into any of the supported forms, send a formData message to the HTTP message. The result is a collection of associations, the same form as the input to the formData: message.

```
(HttpRequest post: 'http://localhost/xx/ValueOfFoo')
    addFormKey: 'foo' value: 'bar';
    addFormKey: 'file'  value: 'myFile';
    formData
```

File entry values are entire message parts, so that all the associated information can be accessed.

```
request := (HttpRequest post: 'http://localhost/xx/ValueOfFoo')
    addFormKey: 'foo' value: 'bar';
    addFormKey: 'text'  filename: 'text.txt' source: 'some text' readStream;
    yourself.
part := request formData last value.
part contents
```

Reading a HTTP response

The response you receive back from a request is structurally very similar to the request. The main difference is that it has a status line rather than a request line.

Accessor methods are provided for extracting information from the response. Accessor methods are defined in HttpResponse and its superclasses.

The most important information is typically in the body. Your application will use information in the header to determine what kind of data is contained in the body, so it can process that data appropriately.

For this section, we'll use the response received from Yahoo:

```
| client req resp |
client := HttpClient new.
req := HttpRequest get: 'http://www.yahoo.com'.
resp := client executeRequest: req.
```

Status line

The status line tells you the HTTP version, and the success status of the request. To get the status line, send statusLine to the response:

```
resp statusLine.
```

which, for the successful request just made, returns:

```
HTTP/1.0 200 OK
```

To get just the parts of this line you can send other, more specific messages. The following messages are useful:

statusLine
Returns the entire response status line (an HttpResponseStatusLine).

protocol
Returns the protocol part of the status line as a String ('HTTP')

version
Returns the protocol version as a String ('1.0')

descriptionString
Returns the response description as a String ('200 OK')

code
Returns only the message code, as a String ('200')

Header fields

The HttpResponse includes a set of header fields and values. Most of the headers are standard fields, and are supported by specific accessor methods. Other special headers may exist as well, and can be accessed using more general methods.

Standard headers

Accessor methods for most standard HTTP header fields are defined in HttpResponse and its superclasses, in the accessing fields message categories.

For example:

connection
Returns a List of **`connection`** parameters, indicating whether the connection is closed (**`close`**) or kept alive (**`keep-alive`**).

contentLength
Returns the value of the **`content-length`** HTTP header field.

date
Returns the **`date`**, as a Timestamp, that the message originated.

Browse the classes for additional field messages.

Additional headers

In addition to the standard header fields, special fields are frequently defined.

If you know the field name, you can get the value by sending a fieldAt: or fieldsAt: message:

fieldAt: *aString*
Returns the first HeaderField named *aString*.

fieldsAt: *aString*
Returns a List of HeaderFields named *aString*.

For example, you can retrieve a **`set-cookie`** field value like this:

```
cookieField := resp fieldAt: 'set-cookie'.
```

However, since more than one set-cookie field may be present, use:

```
cookieFieldList := resp fieldsAt: ‘set-cookie’.
```

It may also be useful in some situations to process fields even though you cannot anticipate their names. To get the collection of all headers, send a header message to the response:

```
headers := resp header.
```

This returns a MessageHeader instance, which contains a List of HeaderFields as its value. Send a value message to this to get the List.

Message body

The point of many of the headers is to give your application the information it needs to process the message body.

In the case of a message with a simple body, you can simply send a body message to the response:

```
msgBody := resp body.
```

If the body is a multipart body, send a parts message to the response:

```
msgParts := resp parts.
```

This returns an OrderedCollection of MimeEntity instances.

How to parse the body is beyond the scope of this section. The most common body contents is a web page, in HTML, in which case the body contents needs to be passed to an HTML parser.

Cookie Support

Cookies are used to introduce state into a series of HTTP exchanges, providing a way to define a “session.” Information is stored on the client system, and then used during the session to identify the client and various aspects of the session.

VisualWorks client support includes support for handling and caching HTTP Cookies. Two specifications are supported: the original Netscape specification, and the RFC2965 specification. Although the Netscape specification was preliminary, and has been subject to criticism, Netscape-style cookies remain the most commonly used, and is the only style supported by most web browsers at this time.

How Cookies are Used During a Session

When an HTTP client makes a request of a server, it respects a one-time response; state is not required for such a simple exchange.

If a server wants to begin a series of exchanges in which state is relevant, it responds to the client request by including an additional response header to the client defining the cookie; either a Set-Cookie (Netscape) or Set-Cookie2 (RFC2965) header.

If the client chooses to continue the exchange, it returns a Cookie header containing the cookie send from the server. The server may either ignore that header, or return a Set-Cookie (or Set-Cookie2) response header with the same or different information as in the Cookie header sent by the client. This exchange of headers continues through the duration of the exchange.

RFC2965 refers to this exchange as a “session.” The session continues until the server effectively ends it by sending a Set-Cookie2 header with MaxAge=0.

How Cookies are Handled

Most of the operations of receiving and caching cookies, and attaching them to outgoing HTTP requests, are handled automatically by HttpClient, depending on the settings for cookie handling.

When cookie processing is enabled (either by default, as described in ”Cookie Handling Settings,” or for an instance as described below), cookie processing is performed automatically for HTTP requests and responses. If an HttpResponse includes a Set-Cookie or Set-Cookie2 header field, a CookieAgent is created to process and create the cookie. The agent holds the cookie in its cache for use in further interactions with the site. Additional requests to the site include the cookie header field for client identification. No further intervention is required.

Cookies are not cached by default, but caching can be enabled either in the Settings Tool or by command (see below). When caching is enabled, cookies held by a CookieAgent are written to the cache, CookieAgent.Registry, when the Http session is closed. No additional interaction is required.

Accordingly, in general there is no need to interact with the CookieAgent. However, if there is reason to do so, the cookie agent can be retrieved from the HttpClient.

Cookie Handling Settings

The API methods described above (see “Default settings” on page 4-1) and the **Http Cookie** page in the Settings Tool provide default settings for cookie handling. The setting options are explained in the online help for the settings pages.

Some of these default settings can be overridden for an HttpClient instance. Some settings are applied directly to the client but most are sent to the cookie agent.

For example, to enable cookie processing, overriding the default to not process cookies, you can write:

```
client := HttpClient new.
client enableCookies: true
```

Other settings can be overridden using class methods defined in CookieAgent. For example, your code might change the cache cookies setting for a specific HTTP client, which it can do only after a cookie has been received and a cookie agent created:

```
client := HttpClient new.
client enableCookies: true.
"Access a site that sends a cookie"
response := client get: 'http://www.amazon.com'.
agent := client cookieAgent.
agent cacheCookies: true.
client close.
```

Note that the cookie is cached only once the client is closed.

Messages that are sent to the HttpClient are:

enableCookies: *aBoolean*
If set to true, enables processing of cookies received.

Messages that are sent to the CookieAgent are:

cacheCookies: *aBoolean*
When enabled (true), cookies that are received during an HTTP session are written to the cookie cache, CookieAgent.Registry, when the HttpClient is closed.

enableLimits: *aBoolean*
When enabled (true), cookie verification limits are checked.

useCachedCookies: *aBoolean*
When enabled (true), cached cookies are used for the session.

Setting Cookie Fields

CookieAgent creates cookies based on Set-Cookie or Set-Cookie2 response headers, and returns the cookie with subsequent responses. This is done automatically, based on cookie handling settings, and there is no need to intervene in the process. Neither is it generally desirable to do so.

However, in case there were a reason to construct your own cookie, the API is provided to do so, for either Netscape or RFC2965 style cookies. Examples in both styles are provided in the class comment for CookieAgent, which you should review.

One special case for which you do need to create a Cookie2 header is the case where the client wants to negotiate up to using RFC style cookies. As described in the RFC, if a client receives a Set-Cookie response header, i.e., in Netscape style, instead of a Set-Cookie2 response header, it should respond with a Cookie header in the Netscape style, but should also send a Cookie2 request header:

```
Cookie2: $Version="1"
```

This informs the server that the client understands the RFC style cookies, and the session could continue in that style. If the server does not understand the RFC style cookies, it will simply ignore the header.

To add the Cookie2 field, create a request for the response, then add the field using addField:. For example,

```
request := (HttpRequest post: 'http://www.amazon.com')
     addField: Cookie2Field version: 1.
```

And then send the request:

```
client executeRequest: request
```

Streaming on an HTTP connection

Streaming support is implemented using layers of stream wrappers to handle different aspects of message transport processing. Aspects like compression, character encoding, HTTP chunking, etc., are processed by their own wrappers.

Basic read protocol

A simple interface for creating a Stream on a URI is provided by these two message selectors:

readStreamCmd: *commandString* **url:** *urlOrString*
Connect, send a request, and answer a http connection stream.

readStreamCmd: *commandString* **url:** *urlOrString* **do:** *aBlock*
Connect, send a request, evaluate *aBlock* on the connection stream, ensure disconnect, and answer the block evaluation result.

In these messages, *commandString* is simply the HTTP command ('GET', 'PUT', 'POST', etc.). The URI must be a complete path, including the protocol and server. The server part is used to make the connection. The remainder of the path is used with the command string to place the command.

readStreamCmd:url: creates an open ReadStream, and holds the stream open for reading using the standard Stream operations. You are responsible for closing the connection when you are finished processing the Stream.

```
| client stream |
client := HttpClient new.
stream := client readStreamCmd: 'GET' url: 'http://www.yahoo.com'.
[ stream atEnd ] whileFalse: [ Transcript show: stream nextLine; cr ].
stream close
```

readStreamCmd:url: creates the ReadStream and performs the actions described in *aBlock* on the stream. *aBlock* is expected to be a one-argument block, with the stream as the input argument. This method ensures that the stream is closed before completing.

```
| client stream |
client := HttpClient new.
stream := client readStreamCmd: 'GET'
    url: 'http://www.yahoo.com'
    do: [ :x | [x atEnd] whileFalse: [Transcript show: x nextLine; cr.] ]
```

Handling attachments

As long as attachments are small, they can be handled in memory. Larger attachments, however, need to be read from and written to files.

Received attachments are written to a file, by default. To toggle this to receive attachments into memory, send

```
HttpBuildHandler saveAttachmentsAsFiles: false
```

Attachment files are saved in a directory, which is by default named **`http-temp-files`** and located in the image directory. To change the default directory use the following expression:

```
HttpBuildHandler defaultAttachmentDirectory: 'myDirectory'.
```

HttpBuildHandler creates a file name based on the **`Content-Disposition`** filename parameter. If a file with this name already exists a new name will be generated. In all cases the framework raises a notification, AttachmentFilename, allowing the user to override the file name on the fly. For example:

```
[ response := client executeRequest: request
    ]   on: AttachmentFilename
        do: [ :ex | ex resume: 'http-temp-files\MyTempFile.txt' ].
```

Chunking Large Attachments

For uploading big files, messages are chunked. HTTP chunking is a transport encoding which splits the message in a number of smaller “chunks” and writes each chunk in sequence. A size indicator is included for individual chunks rather than for the entire message. Writing even very large messages is efficient, because we don’t have to compute the actual byte size of the entire message up front.

When writing a message, the stack of stream wrappers includes a ChunkedWriteStream, which collects the message body bytes in a buffer until the buffer is filled. The buffer size is settable, with the default size set to 4K. If the message body fits entirely into a single buffer, the message is not chunked, but is sent intact, and the message header will include the “Content-length” field with corresponding byte size value. If the body is longer than the buffer, then when the ChunkedWriteStream is about to write the first chunk into the underlying stream, the higher levels of the framework add a "transfer-encoding: chunked" header field and the body is written out in the chunked format.

As was mentioned, the chunk size is settable. To set the size globally (in bytes) send a message:

```
ChunkedWriteStream class>>defaultWriteLimit: aNumber
```

To set the chunk size individually for each message use the following pattern:

```
pwriter := HttpWriteHandler new.
writer chunkSize: aNumber.
writer writeMessage: anHttpMessage on: aWriteStream.
```

or

```
client := HttpClient new.
client chunkSize: 200.
response := client executeRequest: aHttpRequest.
```

Chunking can cause trouble for version 1.0 HTTP servers that do not support it. To suppress chunking, either set the default to not chunk:

```
HttpWriteHandler shouldChunk: false
```

or send doNotChunk to the HttpRequest:

```
Sending not chunked message:
    request := HttpRequest get: url.
    request doNotChunk.
    response := request execute.
```

Messages are chunked by default (shouldChunk: true). The option setting works as follows:

- When shouldChunk is true (default),
    - if the message body size exceeds the size specified by the chunkSize option (defaultChunkSize is set to 4K), the message is chunked;
    - if message body size has fewer bytes than the specified chunkSize, the messages is not chunked, and uses the content-length header instead.
- When shouldChunk to false, the message is not chunked regardless of body size and uses the content-length header instead.

Note that, when shouldChunk is false, the writer needs to be able to determine the exact, final byte size of the message body (e.g., if the body is to be compressed it has to be the compressed size). The size has to be known before it starts writing the body, so that it can inject the correct content-length field into the header. In general in this mode the body is first written into an internal stream to determine the correct byte count, then the header is finished with the right content-length, and finally the body bytes are copied from the internal stream. As an optimization, if the body is simple (i.e. not multi-part) and the size of the body is known in advance, the writer will use that body size for the content-length field and then write the body bytes to the outging stream directly. This does not change the behavior in any way, but does make the handling of this particular case more efficient manner than the other non-chunked cases.

For code samples on how to control chunking in HTTP see the class comment of HttpWriteHandler.

Compression

The streaming implementation also includes an option that allows you to control compression of the outgoing messages. Setting the option to true will transfer the message in gzip format ("Transfer-Encoding: gzip"). The default value is false. Use the following pattern to enable gzip compression of messages:

```
client := HttpClient new.
client useGZipTransfer: true.
[ response := client executeRequest: request ] ensure: [ client close ]
```

or:

```
writer := HttpWriteHandler new.
writer useGZipTransfer: true.
writer writeMessage: httpMessage on: stream.
```

A compressed, chunked message will be sent out using both of these transfer encodings and will include the following header fields:

```
Transfer-Encoding: gzip
Transfer-Encoding: chunked
```

The body will be compressed and then split into chunks of the specified size.

Authentication

In general, HTTP requests are passed without user authorization information (user ID and password). If a 401, “authentication required,” response is received back, the HttpClient creates an instance of AuthenticationPolicy. The class AuthenticationPolicy provides different types of authentication for HTTP messages.

The policy selects a supported authentication scheme from the server challenge, creates an instance of the specific authentication, and adds an authorization field to a request.

The policy will try to handle the server challenge if:

- a user name and password is provided
- the server challenge includes an authentication scheme that is supported by HttpClient

Currently supported authentication schemes are: Basic, Digest, and NTLM. The client side preferences for authentication mechanism are controlled by the authentication order (#authOrder), which can be specified either at the global level (class side) or at the individual instance level.

An HTTP client always sends an unauthorized message first, which results in a challenge response if the site requires authentication/authorization. How the challenge is answered depends on the information available to the HttpClient. Here are a few cases.

- The user name and password provided before the request, by being set in the HttpClient instance.

  ```
  cl := HttpClient new.
  cl username: 'winUsername' password: 'winPass'.
  reply := cl get: aURI.
  ```

  The authentication scheme will be selected from the server 401/407 reply. The user name and password will then be encoded based on this scheme and the request will be sent to the server.

- The HttpClient may be configured to respond with a specific authorization scheme:

  ```
  cl := HttpClient new.
  cl username: 'winUsername' password: 'winPass'.
  cl useBasicAuth.
  reply := cl get: aURI
  ```

  If the specified scheme is not acceptable to the server, the request will fail. In addition to useBasicAuth, used above, there is also useNTLMAuth to specify the policy.

- If the user name and password are not provided, the HttpClient will raise an HttpUnauthorizedError exception. Your error handling code will specify the handling.

  ```
  cl := HttpClient new.
  [reply := cl get: aURI.
  ] on: Net.HttpUnauthorizedError
  do: [ :ex |
      cl username: 'winUsername' password: 'winPass'.
      ex retry]
  ```

- You can also specifying authorization information for a proxy server.

  ```
  proxy := (HostSpec new
              name: 'ntlmAuthProxyServer';
              type: 'http';
              yourself).
  proxy netUser:
      (NetUser username: 'winUsername' password: 'winPass').
  cl := HttpClient new.
  cl
      proxyHost: proxy;
      useProxy: true.
  reply := cl get: html.
  ```

This next sample demonstrates how an HTTPClient uses the Authentication policy.

The HttpClient received the 401 reply from the server. The server can accept the NTLM and Basic authentication.

```
request := HttpRequest readFrom:
'GET http://www.cincomx.com/en/index.asp HTTP/1.1
Host: www.cincom.com:4545
Connection: Keep-Alive' readStream.

reply := HttpResponse readFrom:
'HTTP/1.1 401 Unauthorized
WWW-Authenticate: Negotiate
WWW-Authenticate: NTLM
WWW-Authenticate: Basic realm="testrealm@host.com"' readStream.
```

To process the 401 message the HttpClient creates an instance of the authentication policy.

```
polBuilder := AuthenticationPolicy new.
```

The default policy order is set as

```
AuthenticationPolicy class>>defaultAuthOrder
    ^Array
        with: NTLMAuthentication
        with: DigestAuthentication
        with: BasicAuthentication
```

This order can be changed at the instance level, if desired:

```
polBuilder policiesOrder: (Array
        with: BasicAuthentication
        with: DigestAuthentication
        with: NTLMAuthentication).
```

Set the authentication information:

```
polBuilder username: 'aUser' password:  'password'.
```

Check if the policy can process the reply challenge.

```
polBuilder acceptChallenge: reply request: request.
```

Based on the authentication policy order the Basic scheme will be selected to authorize the request.

```
polBuilder addAuthorizationTo: request.
```

Using a Proxy Server

If you access the internet through a proxy, you should set the proxy information using the Net Settings tool (see NetClient Settings).

For an application environment, you may need to set the proxy programmatically, to provide your own access to the settings. You can do this using either of these HttpClient class methods:

proxyHost: *aString*
 Sets the HTTP proxy to host *aString* on port 80.

proxyHost: *aString* **port:** *anInteger*
 Sets the HTTP proxy to host *aString* on port *anInteger*.

proxyHost: *aString* **port:** *anInteger* **userid:** *aUserName* **password:** *aPassword*
 Sets the HTTP proxy to host *aString* on port *anInteger*, and assigns *aUserName* as the proxy login name. If *aUserName* is not already a defined identity, then create it, and assign *aPassword*.

If you specify a proxy host, you should send the an enableProxy message, to ensure that it is used.

With the proxy specified, you send an HTTP request as usual:

```
| client req resp |
HttpClient proxyHost: 'PROXYSERVER' port: 80.
HttpClient enableProxy.
client := HttpClient new.
req := HttpRequest get: 'http://finance.yahoo.com'.
^resp := client executeRequest req.
```

The request will be sent to the proxy, which forwards the request, receives the response, and passes it to the client.

For proxy authentication, see Authentication above.

HTTP Exception Handling

HttpException and its subclasses are raised as exceptions in response to errors and exceptions in an HTTP exchange.

Subclasses and superclasses are arranged by specificity. HttpException is the most general HTTP error, and so is the super class of them all. HttpClientError is one subclass of HttpException, and applies to all 400 series errors. HttpUnauthorizedError applies specifically to 401 errors, and is a subclass of HttpException.

The most specific error applicable is raised, allowing your application to take specific actions where appropriate. For example, HttpMovedError is more specific than HttpRedirectionError, and both are more specific than HttpException.

Number	Exception Class
1xx	HttpInformationalError
301	HttpMovedError
302	HttpMovedError
3xx	HttpRedirectionError
401	HttpUnauthorizedError
407	HttpProxyAuthenticationError
4xx	HttpClientError
510	HttpNotExtendedError
5xx	HttpServerError
xxx	HttpException

Special handling is built into the HttpClient to handle certain exception cases. For example, authorization errors 401 and 407 initiate retries, if an appropriate user ID and password are identified. (See Authentication for more information.) Similarly, a 301 or 302 error initiates a retry with the new URL, if possible. In such cases, your error processing only needs to handle cases that the built-in handling does not handle.

5

Email

NetClients provides support for three standard email protocols: SMTP for outgoing mail, POP3 for incoming mail, and IMAP for full control over mailbox contents on a server.

Mail clients send and receive mail messages that are created and formatted as described in Chapter 3, “Internet Messages and MIME Types.” This chapter only deals with the mail system protocol itself.

In building a mail client, you typically need to provide for both sending and receiving mail. POP3 and IMAP are mail receiver clients, and SMTP is a sender client. The mail protocols you choose is determined by the support provided by the server you need to access. For internet ISPs, combining POP3 and SMTP is the most common.

For POP3 and SMTP servers that support secure connections using SSL, these client libraries provide secure connection options.

Mailboxes

The Mailbox class and its subclasses provide a high-level interface that makes it simple to check mailboxes. There’s a class for IMAP and for POP3, in this hierarchy:

```
Mailbox
    IMAPMailbox
    POP3Mailbox
```

The following protocol is useful for checking a mailbox:

allHeaders
: Returns a collection of message headers

anyNewMail
: Returns a Boolean; true if there are new messages, false otherwise.

messageCount
Returns the number of messages in the mailbox.

getMessage: *anInteger*
Returns the message, in an instance of LetterInTransit, specified by *anInteger*.

You create either an IMAPMailbox or a POP3Mailbox, and then you can send any of these messages. For example:

```
mailbox := IMAPMailbox           "or POP3Mailbox"
        user: (NetUser username: 'nicki' password: 'hox3')
        server: 'mail.northpole.net'.
     mailbox allHeaders.
```

If there's already an open connection, it is used for the command and left open. If there isn't already a connection, then one is created, and closed upon command completion.

SMTP

SMTP (Simple Mail Transfer Protocol) is the simplest of the mail protocols. It is designed to send an e-mail message from the client to a SMTP server by simply writing a Stream of data to port 25.

SMTP client support is provided in a single class, SMTPClient, which inherits from NetClient.

Creating a SMTP client instance

SimpleSMTPClient instance creation protocol is inherited from NetClient. Since SMTP typically uses port 25, you generally only need to specify the host as a String:

```
SMTPClient host: 'smtp.some_isp.net'
```

If an SMTP client is defined as the default outgoing mail client in the Network Settings, you can access it by:

```
SMTPClient defaultOutgoingMailClient
```

Sending a message

There are two utility messages in the SMTPClient protocol for sending mail messages. To send a single message on a connection, use:

sendMessage: *aMailMessage*
Sends *aMailMessage* to the SMTP server.

This message does the following:

1. Open connection
2. Login
3. Send the message
4. Quit and disconnect

To send multiple messages over a single connection, use:

sendMessage: *aMailMessage*
This message can be used to send a few mail messages over the same connection. The client has to be connected and authenticated before the msesage is sent. After the message is sent the client changes the state from transaction to authenticated. After all messages have been sent, the connection must be closed by sending a quit message.

Depending on the useAuthentication setting, the utilities will start the communication session with the EHLO or HELO commands. Based on the server reply for the EHLO command the client selects a supported authentication scheme and sends authorization information to the server. If the hand shake is successful, the client is moved to the “authenticated” state. The useAuthentication option is set to true by default.

Examples

- Sending an authenticated message over a regular connection:

```
smtpClient := SMTPClient host: 'smtp.somehost.com'.
smtpClient user: (NetUser username: 'username' password: 'password' ).
smtpClient send: message.
```

- Sending an authenticated message over a SSL connection:

```
smtpClient := SMTPClient host: 'smtp.somehost.com'.
smtpClient user: (NetUser username: 'username' password: 'password' ).
smtpClient useSecureConnection.
smtpClient send: message
```

- Sending a non-authenticated message over a regular connection:

```
smtpClient := SMTPClient host: 'smtp.somehost.com'.
smtpClient  useAuthentication: false.
smtpClient send: message
```

- Sending a few messages

```
smtpClient := SMTPClient host: 'smtp.somehost.com'.
smtpClient user: (NetUser username: 'username' password: 'password' ).
smtpClient connect.
[smtpClient login..
    smtpClient send: message1.
    smtpClient send: message2.
    ] ensure:  [smtpClient  quit ]
```

Secure SMTP

To establish a secure SMTP session, set the client to secure mode by sending useSecureConnection to the client.

```
client := SMTPClient new.
client user: (NetUser username: 'username' password: 'password').
client hostName: 'smtp.com' portNumber: '9090'.
client useSecureConnection.
[ client connect ] on: Security.X509.SSLBadCertificate do: [ :ex | ex proceed].
client send: self message
```

You should also handle the SSLBadCerticate exception, as shown.

To set the connection back to normal mode, send useDefaultConnection to the client.

Handling SMTP exceptions

SMTPClient raises no exceptions of its own. General SocketAccessor errors can occur, however, and should be trapped. See “Socket Programming” in *Basic Libraries* for information.

POP3

POP3 (Post Office Protocol 3) is a simple mail-reading protocol that can run over TCP/IP. POP3 enables reading and deleting of messages in a mailbox.

POP3 support is provided by the POP3Client class and several support classes. POP3Client provides the public interface; the remaining classes should be considered private to the implementation.

This section discusses using the POP3Client to receive messages and manage the mailbox. For information about handling messages once they are received, refer to Chapter 3, “Internet Messages and MIME Types.”

Creating a POP3 client instance

POP3Client instance creation protocol is inherited from NetClient. Since POP3 typically uses port 110, you generally only need to specify the host as a String, without specifying the port:

```
POP3Client host: 'pop.some_isp.net'
```

If the POP3 client is defined as the default incoming mail client in the Network Settings, you can access it by:

```
POP3Client defaultIncomingMailClient
```

Note that the default client might be an IMAP client, so this will require some error handling.

Running a POP3 session

A POP3 session begins when a client connects to the server on port 110. The client then waits for the server to send a greeting. Once the greeting is received, the client and server begin exchanging commands and responses. The commands that may be issued are dependent upon the client state, either POP3AuthorizationState or POP3TransactionState. At the end of the exchange, the client closes the connection.

Connecting and disconnecting

For POP3Client, the connection is controlled by these messages, which are sent to a POP3Client instance:

connect
Creates a connection to the server identified by the client and waits for the greeting response from the server.

disconnect
Closes the connection to the server, but does not terminating the session. To complete a transaction, be sure to send quit before disconnect.

quit
Quits the TRANSACTION state, entering the UPDATE state. This does not disconnect from the server.

close
Performs quit followed by disconnect.

By default, POP3Client performs five retries before failing, to accommodate a temporary failure due to a busy server. If the connection cannot be established, a NetClientError is raised.

To end the session, send close to the client to close the connection.

Logging into the server

Once a connection is established, the client enters the POP3AuthorizationState, in which it can send login messages. The client must have a user, which is identified by sending it a user: message with a NetUser instance:

```
pop3Client user: (NetUser username: ‘myUserId’ password: ‘secret’).
```

With the user identified, the client can log in by sending the login message:

```
pop3Client login.
```

Upon successfully logging in, the client state is changed to POP3TransactionState, and mail management commands can be sent.

POP3Client does not default to using the default user set in Network Settings. You can, however, access and use the default user ID:

```
pop3Client user: NetUser registry defaultIdentity.
pop3Client login.
```

Sending POP3 commands

Once you are logged in, issuing POP3 commands is simply a matter of sending the correct command message with any required parameters. Browse the **commands** method category to see the complete set.

Secure POP3

To establish a secure POP3 session, set the client to secure mode by sending useSecureConnection to the client.

```
pop3Client := POP3Client new.
pop3Client user: (NetUser username: 'username' password: 'password' ).
pop3Client hostName: 'hostname'.
pop3Client useSecureConnection.
[pop3Client connect] on: Security.X509.SSLBadCertificate do:
[ :ex | ex proceed].
[pop3Client login; list ] ensure: [pop3Client close ]
```

You should also handle the SSLBadCerticate exception, as shown.

To set the connection back to normal mode, send useDefaultConnection to the client.

POP3Client Commands

POP3 commands defined in RFC 1939 are implemented as POP3Client methods.

The available methods are:

apop: *mailboxNameString* **digest:** *MD5DigestString*
: Provides an alternative login method for POP3 servers that support the POP3 APOP command.

delete: *messageNumber*
: Marks message *messageNumber* to be deleted from the mailbox when it is updated upon disconnect.

deleteMessageIds: *aCollectionOfUIDs*
: Marks all messages in *aCollectionOfUIDs*, each member of which is a unique-ID (see retrieveMessageID: below), for deletion from the mailbox when it is updated upon disconnect.

list
: Returns an OrderedCollection of POP3Status instances, one for each message in the mailbox. Each POP3Status indicates the message number and the size of the message, in octets.

list: *messageNumber*
: Returns a POP3Status for *messageNumber*, containing the message number and its size, in octets.

noop
: Sends a POP3 NOOP (no op) message. The server returns a positive response, if it is operating properly.

reset
: Restores any messages marked for deletion.

retrieveMessage: *messageNumber*
: Returns message *messageNumber* as a ByteString.

retrieveMessageID: *messageNumber*
: Returns the unique ID for *messageNumber* as a ByteString.

retrieveMessageLines: *messageNumber*
: Returns an OrderedCollection of the lines of message *messageNumber*, each line as a ByteString.

status
: Returns a POP3Status containing the number of messages and the total size, in octets.

top: *numberOfLines* **of:** *messageNumber*
: Returns the message header and first*numberOfLines* lines of message *messageNumber*.

Get inbox information

The status, list, and list: messages return information about the inbox in instances of POP3Status. status returns a POP3Status indicating the number of messages in the inbox and the total size of the contents of the inbox.

list returns an OrderedCollection of POP3Status instances, each of which contains a message number and the size of the message. list: returns a single POP3Status containing the message number and size of the specified message.

Each POP3Status responds to the messages messages and octets. messages returns either the total number of messages (status) or the message number (list and list:). octets returns either the total size of messages in the inbox (status) or the size of each message (list and list:).

For example, to print a listing of message numbers and sizes to the Transcript, do:

```
| popClient msgList |
    popClient := POP3Client host: 'pop.yourserver.net'.
    popClient user: (NetUser username: 'jdoe'
        password: 'passwordforjoe').
    popClient connect;
        login.
    popClient list do:
        [ :msg | Transcript show: msg messages printString;
            nextPut: Tab;
            show: msg octets; cr ].
    popClient close. " Sign off and disconnect"
```

Retrieve a message

The retrieveMessage: and retrieveMessageLines: are the most common messages for retrieving a message from the inbox. For both, you specify the message to retrieve by its message number. retrieveMessage: returns the message as a ByteString, and retrieveMessageLines: returns the message as an OrderedCollection of ByteStrings, each ByteString containing one line of the message.

For some purposes the ByteString might be adequate, but for MIME enabled mail clients the message needs to be rendered as a MailMessage. To do this, open a ReadStream on the message ByteString and send a readFrom: message to MailMessage with the read stream as argument, as shown below.

```
| popClient message mimeMsg |
    popClient := POP3Client host: 'pop.yourserver.net'.
    popClient user: (NetUser username: 'jdoe'
        password: 'passwordforjoe').
    popClient connect;
        login.
    message := popClient retrieveMessage: 1.
    mimeMsg := MailMessage readFrom: message readStream.
    popClient close.
```

Delete a message

The delete: and deleteMessageIds: mark messages in the inbox for deletion. Messages are not actually deleted until the session enters the UPDATE state, which occurs when the client is sent quit.

Message numbers are only valid message identifiers during the session that reported the number, so is only reliable during the current session. So, for example, to delete the last message by its message number, use delete: as follows:

```
| popClient lastMsgNum |
    popClient := POP3Client host: 'pop.yourserver.net'.
    popClient user: (NetUser username: 'jdoe'
        password: 'passwordforjoe').
    popClient connect;
        login.
    lastMsgNum := popClient status messages.
    popClient delete: lastMsgNum.
    popClient close.
```

An alternative is to use the message’s unique-ID, which is a server-assigned identifier that uniquely identifies the message across sessions. So, a client application could store message information locally and permit deleting a message at a later time by using this unique-ID, rather than trying to identify the message by its current number.

To get the unique-ID from the mail server, send a retrieveMessageID: message. The collected IDs can then be passed as a collection to deleteMessageIds: for deletion. This example collects all the current message IDs and then deletes all messages using those IDs.

```
| popClient idCollection numMsgs |
    popClient := POP3Client host: 'pop.yourserver.net'.
    popClient user: (NetUser username: 'jdoe'
        password: 'passwordforjoe').
    popClient connect;
        login.
    numMsgs := popClient status messages.
    numMsgs >= 1 ifTrue:
        [ idCollection := 1 to: numMsgs collect:
            [ :num | popClient retrieveMessageID: num ] ].
    popClient deleteMessageIds: idCollection.
    popClient close.
```

Since messages are only marked for deletion until the session is terminated, you can “undelete” messages during the session, by sending a reset message to the client. This undeletes all messages currently marked for deletion. So, in the last message, we could change our mind and restore all the messages by inserting this before the quit message line:

```
    popClient reset.
```

POP3 states

NetClients supports three session states, represented by three classes: POP3State, POP3AuthorizationState and POP3TransactionState.

State changes

POP3State is the default state, and is assigned to a client when it is first created.

Upon successful connection, the client’s state is changed to POP3AuthorizationState. In this state the login message can be sent to the client, which sends the USER and PASS POP3 commands.

Upon successfully logging in, the client’s state is changed to POP3TransactionState. In this state, the usual command messages can be sent, for retrieving messages or message information, and deleting messages from the mailbox. The disconnect message can also be sent in this state, terminating the session, but should be preceded by quit to cause the update..

When the session is terminated using disconnect, the client’s state is returned to POP3State.

State errors

Attempting to send commands to the client while it is in POP3State will raise a POP3StateError. Similarly, attempting to send the login message while the client is in POP3TransactionState, or sending any of the transaction commands while it is in POP3Authorization state, will also raise POP3StateError.

If the session is terminated by the server, such as when the connection is timed out, the client's state remains in its last state; it is not changed to POP3State. Attempting to send commands while the client is in this condition will either raise a general socket error, a Message Not Understood error, or, if the message is inconsistent with its last state, a POP3StateError.

Handling POP3 exceptions

There are only two exceptions specifically relevant to a POP3 session: NetClientError and POP3StateError. General SocketAccessor errors may also occur, however, and should also be handled (refer to "Socket Programming" in *Basic Libraries* for information).

NetClientError is signaled if connect fails to create a connection. For example, if the host is incorrect, the socket cannot be created, resulting in this error. Your application should trap and handle:

```
[ pop3Client connect ]
    on: NetClientError do:
    [ :ex | ex handleThis ]
```

POP3StateError is raised if a command is sent while the client is in the incorrect state. All commands, except user: and pass:, require the client to be in POP3TransactionState, which is set upon successful connection and unset upon disconnection. While the connection is being established, the state is POP3AuthorizationState, which permits user: and pass: commands.

In general, your application will be so constructed that you will not need to handle state errors.

IMAP

IMAP (Internet Message Access Protocol) allows a mail client to access electronic mail messages that are stored on a server and manage a collection of mailboxes. With mail stored on the server, you can maintain your mailboxes equally from a desktop computer at home, a workstation at the office, and a notebook computer while traveling, without the need to transfer messages or files back and forth between these computers.

Due to the size and complexity of the IMAP specification, this section cannot explain all there is to know about IMAP. For additional information, refer to RFC 2060.

Creating an IMAPClient instance

IMAPClient instance creation protocol is inherited from NetClient. Since IMAP typically uses port 143, you generally only need to specify the host as a String, without specifying the port:

```
IMAPClient host: 'imapHost.someISP.net'
```

If the IMAP client is defined as the default incoming mail client in the Network Settings, you can access it by:

```
IMAPClient defaultIncomingMailClient
```

Note that the default client might be a POP3 client, so this will require some error handling.

Running an IMAP session

Unlike POP3 and SMTP sessions, an IMAP connection is expected to be maintained for a long period of time.

From the time the IMAP connection is established until it disconnects, it passes through a series of states, each of which is represented in VisualWorks by a class (see “IMAPClient states” below).

Connect and Log in

Once you have created an IMAPClient instance, you need to connect to an IMAP server and log in.

To connect, send a connect message to the IMAPClient instance:

```
imapClient connect.
```

Once successfully connected, the client state is changed to IMAPNonAuthenticatedState, and you are ready to log in.

Logging in, as always, requires a user ID, supplied by a NetUser instance. Once the user ID is provided to the client, send the login command:

```
imapClient user: ( NetUser username: 'nicki' password: ‘somepassword' ).
imapClient login.
```

IMAP allows for an alternative authentication mechanism, which may be implemented by an IMAP server. For logging into such a server, use the authenticate: message with the appropriate authentication string as argument.

Mailbox maintenance

Once successfully logged in, the client state advances to IMAPAuthenticatedState. Several mailbox maintenance commands are available at this point, such as commands for creating and deleting mailboxes, listing and examining mailboxes, adding a message to a mailbox, and subscribing and unsubscribing to mailboxes.

Message maintenance

Operations on messages require that a mailbox is selected. Upon selection, the client is placed into a IMAPSelectedState. Commands affecting messages, such as retrieving, moving, or modifying a message, in all of which a message is identified by number, are performed for the selected mailbox. The mailbox remains selected until another mailbox is selected, the mailbox is closed or deleted, or the client is disconnected.

Log out

A session ends when it is disconnected, either by the client or the server. The client disconnects by sending a logout message to the IMAPClient instance. The state is returned to the default IMAPState, and no further commands may be sent.

The server may close the connection by timing out, usually after at least 30 minutes of inactivity. The client state is not updated in this case.

IMAPClient states

In the course of an IMAP session the client enters a variety of states, represented in VisualWorks as the following classes:

IMAPState
The default state for a client prior to connection and following logout. No IMAP commands can be sent the client in this state, though the connect message can be sent.

IMAPNonAuthenticatedState
Upon successful connection the client enters this state, for user authentication. The login message can be sent in this state, to authenticate using a username and plain-text password.

IMAPAuthenticatedState
Upon successful authorization the client enters this state, for selecting a mailbox. A mailbox must be selected, by sending a select:, examine:, or create: message.

IMAPSelectedState
The client is set into this state when a mailbox is selected by sending a select:, examine:, or create: message.

Only certain IMAP commands can be sent to an IMAPClient in any given state. The messages implementing IMAP commands are all defined in IMAPState as returning an error signal. IMAPState subclasses for which a message is valid reimplement the message. Browse the **commands** message category for these classes.

Command responses

In the course of a session, IMAP clients send commands to the server, and the server sends back responses reporting results and status.

Most IMAPClient command messages return an instance of IMAPCommand, which contains the server responses. Utility messages typically return the most relevant part or parts of the response, for easier processing, rather than the IMAPCommand object, and so are not discussed in this section.

One important response is the completion response, which indicates whether the command succeeded or not. Send the message completionResponse to the IMAPCommand to get an IMAPResponseTagged object. The status field, which is accessed by the status message, is frequently useful to verify whether a command has succeeded:

```
( imapClient select: 'some box' )
    completionResponse status = 'OK'
        ifTrue: [ "do some operation on messages" ]
```

Typical completion responses are: 'OK' for success, 'NO' for failure, and 'BAD' for an incorrect command or arguments. Even better for testing for success or failure, send a successful or failed message, which simplifies the above to:

```
( imapClient select: 'some box' ) successful
        ifTrue: [ do some operation on messages ]
```

Command messages also return a command response, which can be accessed by sending a commandResponse message to the IMAPCommand. Depending on the command, the response may be a IMAPResponse or an undefined object. The contents of the IMAPResponse also varies, depending on the command, and so needs to be taken into account. For example, a status: message

```
(client status: 'blapperty'
    criteria: #('unseen' 'uidnext' 'messages') ) completionResponse
```

returns an IMAPResponse with a 2-element Array in its parameters variable. The second element contains the information you want in a collection of field names and values, which you would access like this:

```
(client status: 'blapperty'
     criteria: #('unseen' 'uidnext' 'messages') )
         completionResponse parameters at: 2.
```

A fetch: command, on the other hand, returns a single element in its parameters Array, which itself contains a collection. For example,

```
(client fetch: #(1 2 3) retrieve: #(uid rfc822) )
     commandResponse parameters
```

returns a single-element Array in parameters, which in turn contains a 4-element Array. The first and third elements are the names of the requested fields ('UID' and 'RFC822', respectively), and the second and fourth elements are the respective field values.

In other cases, such as a select: command:

```
(client select: 'blapperty') commandResponse
```

the value is nil, and so not interesting.

All IMAPCommands include server responses in their response variable, which holds an OrderedCollection of server responses. If you require server responses from, for example, the select: command, you can get them by sending a responses message to the IMAPCommand. Since it's an ordered collection and the order is consistent on the server, you can extract the information you want. For example to get the UIDVALITITY response, send:

```
( imapClient select: 'bloopity' ) responses at: 5
```

The response is an IMAPResponseStatus whose value is an Association, whose value is the UIDVALIDITY value, so you can dig further with:

```
( ( imapClient select: 'bloopity' ) responses at: 5 ) value value.
```

Browse the various classes mentioned above to find additional methods for accessing response values.

Message flags

IMAP supports a variety of message flags that indicate various dispositions of the messages.

The most commonly set flags to set are \Seen, indicating that the message has been read, and \Delete, indicating that the message is to be cleared from the mailbox at the next expunge. These are supported by the following IMAPClient messages:

markAsSeen: *aCollection*
: Set the **`\Seen`** flag for the message numbers listed in *aCollection* in the currently selected mailbox.

markForDelete: *aCollection*

Set the **`\Delete`** flag for the message numbers listed in *aCollection* in the currently selected mailbox. Messages will be deleted at next expunge.

The messages are applied to the currently selected mailbox, so ensure that the correct mailbox is selected before sending the message.

```
imapClient select: 'bloopity'.
imapClient markAsSeen: #( 1 2 5 9 ).
```

To set other flags, send a store: message the client, after selecting the target mailbox. The argument is a String consisting of the message numbers to flag, '+FLAGS ', and the list of flags to set enclosed in parentheses. So, for example, to mark a message (say, number 5) for urgent or special attention, you can set the **`\Flagged`** flag:

```
imapClient select: 'bloopity'.
imapClient store: '5 +FLAGS (\Flagged)'
```

A range of numbers, say 1 to 3, is specified by separating the numbers with a colon: '1:3'. For a list of numbers, separate with commas: '1,3,7'.

To remove a flag, use '-FLAGS' instead of '+FLAGS'. For example, to mark a sequence of messages as unread, send:

```
imapClient select: 'bloopity'.
imapClient store: '1:5 -FLAGS (\Seen)'.
```

Standard IMAP flags that you can set are: **`\Answered`**, **`\Deleted`**, **`\Flagged`**, **`\Seen`**, and **`\Draft`**. Other flags may be defined by your server implementation.

Mailbox names

Mailbox naming conventions are determined by the server implementation, except that there must be one case-insensitive name, “INBOX,” to serve as the main mailbox, so you are always assured to have that mailbox name available.

IMAP mailboxes may be arranged in a hierarchy. If they are, the level separator is a single character specified by the server, such as the forward slash (“/”).

Some IMAP implementations may also support multiple services, and so support Usenet as well as mail. In this case, namespaces are specified. In the hierarchical name, the namespace name is prefixed with “#”, allowing a name such as "#news.comp.mail.misc".

For other naming rules, refer to RFC2060.

Mailbox maintenance

IMAP allows you to perform a variety of maintenance operations on mailboxes. The client needs to be in IMAPAuthenticatedState or IMAPSelectedState to perform these operations, so the client must be logged into the server.

A number of these basic command messages are also supported by utility messages, some of which are mentioned and/or illustrated in examples. Browse the IMAPClient **utility** method category for others.

select: *nameString*
Makes *nameString* the current mailbox, giving read/write access to the messages in it, and sets the client state to IMAPSelectedState. Returns an IMAPCommand with an OrderedCollection of status responses in its responses variable.

examine: *nameString*
Same as select: except that access to the messages is read-only.

create: *nameString*
Creates a new mailbox, *nameString*. If no mailbox was selected, then *mailboxName* is selected as the current mailbox and the client state is set to IMAPSelectedState.

delete: *nameString*
Deletes mailbox *nameString*. If *nameString* is the current mailbox, then the client state is set to IMAPAuthenticatedState.

rename: *nameString* **newname:** *newNameString*
Renames *nameString* mailbox to newN*ameString*. No changes are made to current mailbox selection of client state.

subscribe: *nameString*
Subscribes to the mailbox *nameString*.

unsubscrible: *nameString*
Unsubscribes from the mailbox *nameString*.

list: *namePattern*
list: *aRefString* **mailbox:** *namePattern*
Returns an IMAPCommand with an OrderedCollection in its responses variable, containing mailbox names matching *namePattern*, which may contain pattern matching characters. Asterisk (*) matches any number of characters, percent (%) matches any single character except the hierarchy delimiter. (See the examples below.) If *aRefString* is specified, it is the hierarchy prefix to the *namePattern*.

lsub: *aRefString* **mailbox:** *namePattern*

Returns an IMAPCommand with an OrderedCollection in its responses variable, containing names of subscribed mailboxes matching *namePattern*, relative to *aRefString*, as described for list:.

status: *nameString*
status: *nameString* **criteria:** *aStringOrCollection*

Returns an IMAPCommand containing an OrderedCollection of status information in its responses variable. For status: the number of messages in the mailbox *nameString* is returned. For status:criteria:, the information requested by *aStringOrCollection* is returned. Status items that may be requested are: MESSAGES (number of messages); RECENT (number of messages with **`\Recent`** flag set); UIDNEXT (the next UID value to be set); UIVALIDITY (the UID validity number of the mailbox); UNSEEN (the number of messages that do not have the **`\Seen`** flag set).

append: *messageString* **to:** *nameString*
append: *messageString* **to:** *nameString* **flags:** *flagList* **date:** *aDate*

Adds *messageString*, a RFC822 format message as a String, to mailbox *nameString*. The longer form allows you to also specify message flags, as a String containing a parenthesized list, and a date String, either or both of which may be nil.

Determine the number of messages in the inbox

A common piece of information frequently needed about a mailbox is how many messages there are in it, or how many messages in a specific state are in it.

The simple message count is returned by the messageCount: message, which returns the number of messages in the specified mailbox. If you want the message count for inbox, you can use messageCount message:

```
| imapClient inCount outCount |
imapClient := IMAP Client host: 'somehost.net'.
imapClient user:
    ( NetUser username: 'nicki' password: ‘somepassword').
imapClient login.
"Get inbox count"
inCount := imapClient messageCount.
"Get outbox count"
outCount := imapClient messageCount: 'outbox'.
^Array with: inCount with: outCount.
```

To get the count of only new messages, send newMessages or newMessages:. To get the number of unseen messages in the inbox, send unseenMessages.

The messages mentioned above are utility messages that send a status: message with specific arguments. Refer to “Getting mailbox status information” below for using this message with additional arguments.

Getting mailbox status information

Beyond the message count, there are several status items you can request from a mailbox. Status item names are case-insensitive:

MESSAGES
: Returns the number of messages in the mailbox.

RECENT
: Returns the number of messages with **\Recent** flag set, which is set by the server to indicate that the message is new during the current session. The flag is unset by the server when the client disconnects or the mailbox is closed. The client cannot set this flag.

UIDNEXT
: Returns the next UID value to be assigned to a message in this mailbox. See “Working with unique identifiers” below.

UIDVALIDITY
: Returns the UID validity number of the mailbox. See “Working with unique identifiers” below.

UNSEEN
: The number of messages that do not have the **\Seen** flag set, which is set by the client to indicate that the message has been viewed.

To get the status information, send a status:forMailbox: message with a String or a collection of Strings specifying the status fields, and a mailbox name. This is a utility method that returns a List of the status field values as Strings, in an order determined by the server but is frequently as shown above (not the order requested).

```
| imapClient resp |
imapClient := IMAPClient host: 'somehost.net'.
imapClient user:
    (NetUser username: 'nicki' password: ‘somepassword').
imapClient login.
^resp := imapClient
    status: #( 'recent' 'uidvalidity' 'unseen' 'uidnext' 'messages' )
    forMailbox: 'bloopity'.
```

For example, the above may return List ('2' '0' '8' '106708' '2').

To get more information, which may be necessary, for example, if the order of status fields from your server is not known, you can send a status:criteria: message. Notice that the order of arguments is reversed from the utility method above, though the arguments themselves are the same. This message returns an IMAPCommand instance, from which the responses can be retrieved as follows:

```
imapClient status: 'bloopity'
    criteria: #( 'unseen' 'uidnext' 'messages' 'recent' 'uidvalidity' )
    commandResponse parameters at: 2
```

This command returns an Array containing the same values as above, but each preceded by its status field: #('MESSAGES' '2' 'RECENT' '0' 'UIDNEXT' '8' 'UIDVALIDITY' '106708' 'UNSEEN' '2').

Selecting a mailbox

Before performing maintenance operations on messages, you must select a mailbox. A mailbox can be selected for either read/write access, or for read-only access. To select a mailbox for read/write access, send a select: message to the client with the mailbox name:

```
imapClient select: 'mailbox'.
```

For read-only access, send examine: instead. Both commands return an IMAPCommand with an OrderedCollection in its responses variable that contains information about the mailbox.

A slight simplification is provided by the utility message examineMailbox:.

```
imapClient examineMailbox: 'mailbox'.
```

This message returns the OrderedCollection of responses.

Creating/deleting a mailbox

To create a new mailbox, send a create: command with the mailbox name as a String:

```
imapClient create: 'bloopity'.
```

The mailbox may be in a hierarchy, in which case the mailbox name includes the entire path, starting at the root, with each element in the hierarchy separated by a server assigned separator character (frequently “/”). Any elements in the path that do not already exist are also created.

```
imapClient create: 'bloopity/blip'.
```

To delete a mailbox, send a delete: message with the mailbox name:

```
imapCleint delete: 'bloopity'.
```

Again the name may be a path in the hierarchy, starting at the root. If a mailbox is deleted that contains other mailboxes, all are deleted.

Rename a mailbox

To rename a mailbox, send a rename:newName: message to the client. | client.

```
imapClient rename: 'bloopity' newName: 'blapperty'.
```

Although the mailbox name is changed, its UIDVALIDITY number remains the same, allowing a client to resynchronize with the server.

List mailboxes

To get a list of mailboxes, send a list: message, with a mailbox name pattern String as argument. You can use an asterisk (*) to match any number of characters, or a percent sign (%) to match any single character except the hierarchy divider.

For example, to list all mailboxes, send:

```
imapClient list: '*'
```

The list is returned as an OrderedCollection of IMAPResponse objects in the IMAPCommand responses variable. You can check for the existence of a specific mailbox by listing only it:

```
imapClient list: 'inbox'
```

If the mailbox exists, it is returned in the collection; otherwise, the collection is empty.

To list only mailboxes relative to an initial segment of a mailbox hierarchy, send a list:mailbox: message. The first argument is the reference mailbox, and may be a path fragment starting at the root. The second argument is a pattern, and may also include path information. The reference mailbox name is prepended to the mailbox name. So,

```
client list: 'bloopity/' mailbox: 'bl%p'
```

lists all mailboxes matching 'bloopity/bl%p'.

Add a message to a mailbox

Placing a message in a mailbox is a mailbox maintenance task, and so can be performed while the client is in IMAPAuthorizedState. To place a message in the mailbox, send an append:to: message to the client with the message and the mailbox as Strings.

The message must be represented as a ByteString, so if you construct it as a MIME message, send printString to convert it:

```
message := MailMessage newTextPlain.
message
    from: 'santa@northpole.net';
    to: 'darlene@goodkid.net';
        subject: 'Start making your list now';
        text: 'What would you like for Christmas?'.
imapClient append: message printString to: 'blippy/blop'
```

Message maintenance

The following messages implement the basic IMAP commands for working with mail messages. There are also a variety of utility methods available that simplify certain specific tasks. The following sections discuss how to use the command and utility methods to manage mail messages.

check
: Requests a “checkpoint” of the currently selected mailbox. A checkpoint is any server implementation-dependent housekeeping associated with the mailbox.

close
: Permanently removes any messages marked for deletion in the currently selected mailbox, and returns the client to IMAPAuthenticatedState. No EXPUNGE responses are returned.

expunge
: Permanently removes any messages marked for deletion in the currently selected mailbox. An EXPUNGE response is returned for each message deleted.

search: *aCriteria*
: Returns an IMAPCommand with an OrderedCollection in its responses variable containing the message sequence numbers of all messages in the currently selected box matching *aCriteria*. See the examples below.

fetch: *aCriteria*
fetch: *messageNumbers* **retrieve:** *aCriteria*
: Retrieves message information from the currently selected mailbox, based on *aCriteria*. The second form restricts the command to the specified *messageNumbers*, which is a collection of message numbers. Several utility messages are provided to simplify fetches. See the examples below for more information.

store: *argumentString*

Sets and removes flags from a message. The argumentString consists of a list or range of numbers, '+FLAGS' or '-FLAGS', and a parenthetical list of flags to set or unset. See "Message flags" above.

copy: *msgNumberCollection* **to:** *mailboxName*

Copies the messages listed in *msgNumberCollection* to the mailbox *mailboxName*.

Reading message data

The fetch: and its family read message data from the current mailbox. The basic fetch: command takes a String argument, where the structure of the String is as specified for the FETCH command in RFC 2060, and is quite complex.

For purposes of retrieving a mail message from a mailbox, the fetchMessage: message is considerably simpler. Given a message sequence number, it returns an Association with the message number as key and an Array containing the message in a ByteString as its value.

Since, in general, you will want to handle the message as a MailMessage rather than as a ByteString, the following example illustrates the conversion:

```
| imapClient message msgString |
imapClient := IMAPClient host: 'somehost.net'.
imapClient user:
      (NetUser username: 'nicki' password: ‘somepassword').
imapClient login.
imapClient select: 'inbox'.
msgString := (( imapClient fetchMessage: 1 ) value first ) .
message := MailMessage readFrom: msgString readStream.
^message.
```

The various parts of the message can then be accessed using all the methods provided by MailMessage and its superclasses.

To retrieve multiple messages, use fetchMessages:, with a collection of message sequence numbers. Similarly, to retrieve only the headers of several messages, send fetchMessageHeaders: with a collection of message sequence numbers. Browse the **utility** method category for additional methods.

To more fully specify the message data to be retrieved, send a fetch:with: utility message. The first argument is a collection of message numbers, and the second argument is a collection of fields, typically as Strings. (The method is quite forgiving and accepts unquoted field names, which it renders internally as either Strings or Symbols). For example:

```
imapClient fetch: #( 3 8 ) with: #( 'UID' 'RFC822' ).
imapClient fetch: #(1 4 5 6 ) with: #( UID BODY ).
```

Several fields (defined in RFC2060) can be retrieved this way:

BODY
: Non-extensible form of BODYSTRUCTURE.

BODYSTRUCTURE
: The MIME body structure of the message, computed by the server from the header information.

ENVELOPE
: The envelope structure of the message.

FLAGS
: The flags that are set for the message.

INTERNALDATE
: The internal date of the message.

RFC822
: The entire message.

RFC822.HEADER
: The message header.

RFC822.SIZE
: The message size.

UID
: The message unique identifier.

If you need more complete control over the requested data than this, use the fetch: message, which takes a String in the format specified in RFC2060. This allows you to use macros, such as ALL and BODY.PEEK, as well as including parameters. For example:

```
imapClient fetch: '1:3 (uid Body.Peek[Header.Fields (Message-id From)])' .
```

Copy a message to another mailbox

To copy a message from the current mailbox to another, send a copy:to: message to the IMAPClient instance. The first argument is a collection of message numbers and the second is the target mailbox name as a String.

```
imapClient select: 'inbox'.
imapClient copy: #(1) to: 'Deleted Items'.
```

This is a copy, so the message remains in the original mailbox. To effect a move, such as when deleting an item by moving it to the Deleted Items tray, follow the copy with a delete:, as shown below.

Delete a message

To delete a message from the current mailbox, send a markForDelete: message to the IMAPClient instance. The argument is a collection of message sequence numbers to be deleted.

```
imapClient select: 'inbox'.
imapClient copy: #(1) to: 'Deleted Items'.
imapClient markForDelete: #(1).
```

Note that the message is not actually removed from the mailbox until the session is terminated, you close the mailbox (send close to the client with the mailbox selected), or you send expunge to the client with the mailbox selected.

Search for messages

The searchMessages: message returns an Array of message numbers in the currently selected mailbox matching the search criteria. The search criteria are specified as a String argument. For example,

```
imapClient select: 'inbox'.
msgNumbers := imapClient searchMessages: 'UNSEEN FROM "Fred"'.
```

returns an Array of message numbers in the inbox that have “Fred” in the FROM field, and which have not yet been read. Note that “Fred” is matched as if it were a pattern, *Fred*, matching “Fred Mayor”, “Mayor, Fred”, or “Freda Fredrickson”.

The following items may be included in the search criteria String. For items with a string argument, the string is in double quotation marks (“ “). Dates are given as described in RFC2060, but briefly in the form:

dd-mmm-yyyy hh:mm:ss zone

where *dd* is a one or two-digit day number (e.g., 1 or 21), *mmm* is the first three month name characters (e.g., Jan or Mar), and *yyyy* is the four-digit year number. The time component is hours minutes seconds. The *zone* is the number of hours and minutes relative to Greenwich (e.g., -0700 for PDT). For example:

16-Oct-2001 17:47:32 -0400

message-set

Include messages with these message sequence numbers. Individual numbers are separated by a comma, 2,4,6,8. Ranges are specified by two numbers separated by a colon, 4:8. corresponding to the specified message sequence number set

ALL

All messages in the mailbox.

ANSWERED
Messages with the `\Answered` flag set.

BCC *string*
Messages that contain the specified string in the message's BCC field.

BEFORE *date*
Messages whose internal date is earlier than the specified date.

BODY *string*
Messages that contain the specified string in the body of the message.

CC *string*
Messages that contain the specified string in the envelope structure's CC field.

DELETED
Messages with the `\Deleted` flag set.

DRAFT
Messages with the `\Draft` flag set.

FLAGGED
Messages with the `\Flagged` flag set.

FROM *string*
Messages that contain the specified string in the envelope structure's FROM field.

HEADER *field-name string*
Messages that have a header with the specified header *field-name* with the specified *string* in the field-body.

KEYWORD *flag*
Messages with the specified keyword set.

LARGER *octets*
Messages with a size larger than the specified number of *octets*.

NEW
Messages that have the `\Recent` flag set but not the `\Seen` flag.

NOT *search-key*
Messages that do not match the specified search key.

OLD
Messages that do not have the `\Recent` flag set.

ON *date*
Messages whose internal date is within the specified date.

OR *search-key1 search-key2*
Messages that match either search key.

RECENT
Messages that have the **`\Recent`** flag set.

SEEN
Messages that have the **`\Seen`** flag set.

SENTBEFORE *date*
Messages whose Date: header is earlier than the specified *date*.

SENTON *date*
Messages whose Date: header is within the specified *date*.

SENTSINCE *date*
Messages whose Date: header is within or later than the specified *date*.

SINCE *date*
Messages whose internal date is within or later than the specified *date*.

SMALLER *octets*
Messages with a size smaller than the specified number of *octets*.

SUBJECT *string*
Messages that contain the specified *string* in the envelope structure's SUBJECT field.

TEXT *string*
Messages that contain the specified *string* in the header or body of the message.

TO *string*
Messages that contain the specified *string* in the envelope structure's TO field.

UID *uid-set*
Messages with unique identifiers corresponding to the specified unique identifier set.

UNANSWERED
Messages that do not have the **`\Answered`** flag set.

UNDELETED
Messages that do not have the **`\Deleted`** flag set.

UNDRAFT
Messages that do not have the **`\Draft`** flag set.

UNFLAGGED
Messages that do not have the **`\Flagged`** flag set.

UNKEYWORD *flag*

Messages that do not have the specified keyword set.

UNSEEN

Messages that do not have the **\Seen** flag set.

Here are a couple more examples:

```
imapClient searchMessages: '1:100 BEFORE 1-Feb-1999'.
imapClient searchMessages:
    'FLAGGED SINCE 1-Feb-1999 NOT FROM "Smith"'
```

Setting are reading message flags

Message flags mark messages for various purposes, as described above in "Message flags".

For example, it is the responsibility of the client to mark a message as seen when it has fetched the message. The client can use the presence of the flag to highlight messages that have not been read.

To set a flag for a message in the current mailbox, send a store: message to the IMAPClient with a String argument specifying the message or messages to flag, the flag operation, and the flag to set or unset. For example, to mark a message as read, or seen, send:

```
imapClient select: 'inbox'.
imapClient store: '3 +FLAG SEEN'.
```

To mark it as unread, use -FLAG instead.

To retrieve the flags for a message or a collection of messages, send a fetch:with: message, as described above in "Reading message data". Rather than retrieving the message, though, you just want its flags.

```
imapClient select: 'inbox'.
imapClient fetch: #( 3 ) with: #('FLAGS').
```

Working with unique identifiers

Messages have both sequential numbers and unique identifier numbers within a mailbox. Most commands refer to messages by their sequential numbers, but at times it is better to use their unique identifiers.

The message UID is a 32-bit value guaranteed to be unique within the mailbox. In general, the number persists between sessions, but is guaranteed to remain constant during a session. UIDs are assigned to messages in strictly ascending order.

Each mailboxes has a unique identifier validity number, a 32-bit value, which remains constant even if the mailbox is renamed or moved. This allows a client to resynchronize with the server even after such a change is made. If a mailbox is deleted and recreated, a new UID validity number is assigned.

Combining the message UID and the mailbox UID validity number provides a 64-bit unique identifier for each message.

Three messages are provided specifically for working with message UIDs:

searchMessagesForUids: *aCollection*
: Returns a collection of message sequence numbers corresponding the UIDs listed in *aCollection*. The elements of *aCollection* are Strings representing individual message numbers (e.g., '5') or ranges of message numbers (e.g., '1:5').

uidFor: *aCollection*
: Returns an OrderedCollection of UIDs corresponding to the message sequence numbers listed in *aCollection*.

uid: *commandString*
: Implements the IMAP UID command. *commandString* is an IMAP COPY, FETCH, STORE, or SEARCH command, as described in RFC2060. For COPY, FETCH, and STORE, the message number arguments are interpreted as UIDs. For SEARCH, the message number arguments are sequence numbers, but the numbers returned are UIDs.

Using these commands you can convert between UIDs and sequence numbers as needed. For instance, since UIDs don't change, if your client caches messages you will want to use the UIDs to coordinate actions on the cached copy with the server copy. So, if you have a cached message with a UID of 2245, you can delete it from the server as follows:

```
| msgSeqID |
imapClient select: 'inbox'.
msgSeqID := imapClient searchMessagesForUids: #(2245).
( imapClient copy: msgSeqID to: 'Deleted Items' )
    completionResponse status = 'OK' ifTrue:
        [ imapClient markForDelete: msgSeqID ; expunge ].
```

A mailbox UID validity number is returned as part of the response when the mailbox is selected or examined, or when its status is requested. The status:forMailbox: command is the easiest way to get the UID validity number, which returns a List of the requested values:

```
( imapClient status: #(UIDVALIDITY) forMailbox: 'inbox' ) first.
```

Since only one value was requested, we can be sure it is the first in the list. Use this number to validate a mailbox when synchronizing the client with the server. If the mailbox by this name has been deleted and recreated, the number will be different, indicating the change.

Handling IMAPClient Errors

There are only two exceptions specifically relevant to a IMAP session: NetClientError and IMAPStateError. General SocketAccessor errors may also occur, however, and should also be handled (refer to “Socket Programming” in *Basic Libraries* for information).

NetClientError is signaled if connect fails to create a connection. For example, if the host is incorrect, the socket cannot be created, resulting in this error. Your application should trap and handle:

```
[ imapClient connect ]
    on: NetClientError do:
    [ :ex | ex handleThis ]
```

IMAPStateError is raised if a command is sent while the client is in the incorrect state, as described in “IMAPClient states” above. You can capture the state that originates the error by sending originator to it. For example, if the client is in IMAPAuthenticatedState, this will return the offended state:

```
[ imapClient fetch: #(1) retrieve: #(RFC822) ]
    on: IMAPStateError do: [ :err | ^err originator ].
```

Mail Attachments

Attachments are files included in a message, but typically in a format recognizable to an application other than the reader. The following messages for handling attachments is provided by MailMessage.

addFileAttachment: *aFilename*
Adds a part to the message containing the specified *aFilename*, and encodes it appropriately.

attachments
Returns an OrderedCollection of MimeEntity instances containing the attachments.

attachmentAt: *index*
Returns the MimeEntity containing the attachment at *index*.

attachmentNames
Returns a List of the file names of the attachments.

saveAttachment: *aMimeEntity* **on:** *aStream*
Writes the attachment *aMimeEntity* on *aStream*.

saveAttachmentAt: *index* **on:** *aStream*
Writes the attachment at *index* on *aStream*.

Retrieve the names of attachments

A client typically needs to list attachments in a form allowing the user to select one for processing, such as saving to disk or opening in an application. Send an attachmentNames message to the MailMessage to get a list of names.

```
| message msg |
msg := mailClient retrieveMessage: 4.
message := MailMessage readFrom: msg readStream.
^message attachmentNames.
```

Save an attachment

Mail attachments can be automatically saved to an external file. This feature is configurable by sending a saveAttachmentsAsFiles: message to MimeParserHandler. The default setting for mail attachments is to *not* save attachments (false), which is different from the HTTP default (true).

Mail attachment files are saved in a directory, which is by default named **`mail-temp-files`** and located in the image directory. Use the following expression to change the default directory:

```
MailBuildHandler defaultAttachmentDirectory: 'myDirectory'
```

The file names for attachments are based on the filename parameter in the Content-Disposition header fields. If a file with that name already exists a new name will be generated. Once the file name is determined the framework raises a notification, AttachmentFilename, allowing the application code to override the file name on the fly. If the notification is not handled the originally suggested file name will be used. For example:

```
input :=
    'From: zz@holcim.com
Content-Type: multipart/related;
boundary="--11"

----11
Content-Type: text/plain; name="budd.txt"
Content-Disposition: attachment; filename="budd.txt"
Content-Transfer-Encoding: base64

QWxhZGRpbjpvcGVuIHNlc2FtZQ==

----11--
' readStream.
    [   message := MailBuildHandler new
                removeContentTransferEncoding: true;
                saveAttachmentsAsFiles: true;
                readFrom: input.
    ]   on: AttachmentFilename
        do: [ :ex |
"The suggested a FATFilename('mail-temp-files\budd.txt')  is replaced
by  image\my-budd.txt "
                ex resume: 'my-', ex filename tail ].
    message parts first contents
```

If the file name provided by the AttachmentFilename notification already exists, the framework raises an error, AttachmentFileExists. This error is resumable allowing the user specify a new file name as a resumption parameter. If it is resumed without a parameter or the new filename from the parameter also exists, the corresponding file will be deleted and the name reused for the attachment. For example:

```
input :=
    'From: zz@holcim.com
Content-Type: multipart/related;
    boundary="--11"

----11
Content-Type: text/plain; name="budd.txt"
Content-Disposition: attachment; filename="budd.txt"
Content-Transfer-Encoding: base64

QWxhZGRpbjpvcGVuIHNlc2FtZQ==

----11--
' .
    [   [   message := MailBuildHandler new
                saveAttachmentsAsFiles: true;
                readFrom: input readStream.
        ]   on: AttachmentFilename
            do:  [ :ex | ex resume: 'temp-', ex filename tail ].
    ] on: AttachmentFileExists do: [ :ex | ex resume: 'anotherTemp.txt' ]
```

Obviously, if this exception is not resumed, the attachment will not be saved and parsing of the enclosing message containing this attachment will end here, unfinished.

Decoding an attachment

A mail message attachment might not specify the character set, as in this sample:

```
Content-Type: application/octet-stream
Content-Transfer-Encoding: base64
Content-Disposition: attachment;
    filename="=?windows-1250?B?NCAtIFD47WxpmiCebHWdb3Xoa/0ga/
    nyIPpw7Gwg7+FiZWxza+kg82R5?="

    UPjtbGmaIJ5sdZ1vdehr/SBr+
    fIg+nDsbCDv4WJIbHNr6SDzZHkuIA0KUNjNTEmKII5MVY1PVchL
    3SBL2dIg2IDMTCDPwUJFTFNLySDTRFkuIA==
```

After removing the Content-Transfer-Encoding field the attachment body value will be displayed using the ASCII encoder. To display the attachment using the correct character set, set the default character set:

```
Settings defaultCharsetEncoder: aSymbol.
```

Then send a `decodedContents` message to the mime entity to decode the entity body value.

If defaultCharsetEncoder is set to nil, the default locale character set will be used to decode mime contents.

Mail Archives

Reading mail messages from archive files is processed differently than from a server. One significant difference is that, unlike reading from a mail server using a transient stream, the expectation is that when reading from a file the source will be persistent. Accordingly, it is not necessary to hold the entire message in memory, but can simply point to the source.

Creating a message from an archive file is done by MailFileReader, rather than MailBuildHandler which is used for transient sources. Suppose you have an archived message, and open a readstream on it. which we can represent as (the readstream will be on a file):

```
input :=
	'From: zz@holcim.com
Content-Type: multipart/related;
	boundary="--11"

----11
Content-Type: text/plain; name="budd.txt"
Content-Disposition: attachment; filename="budd.txt"
Content-Transfer-Encoding: base64

QWxhZGRpbjpvcGVuIHNlc2FtZQ==

----11--
' readStream.
```

To create the MailMessage from the archive, simply read from the stream.

```
	message := MailFileReader readFrom: input.
	message parts last contents.
```

Index