
ℵ Programming Language

Library Reference

This documentation is bound to the **Aleph** programming language license and therefore shall be considered free. This documentation can be redistributed and/or modified, providing that the copyright notice is kept intact. This documentation is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose. In no event shall the copyright holder be liable for any direct, indirect, incidental or special damages arising in any way out of the use of this documentation or the software it refers to.

CONTENTS

Preface	ix
The Aleph programming language	ix
Features	ix
Aleph engine	x
Flexible Distribution	x
 License	 xiii
 1 Input/Output Services	 1
1.1 The aleph-sio library	1
1.2 Input and output stream	1
1.2.1 Input stream	1
1.2.2 Output stream	1
1.3 File streams	1
1.3.1 Stream information	2
1.3.2 Reading and writing	2
1.4 Input stream status	3
1.4.1 The valid-p predicate	3
1.4.2 The eof-p predicate	3
1.4.3 The read method	3
1.4.4 Buffer read mode	4
1.5 Multiplexing	4
1.5.1 Selector object	4
1.5.2 Waiting for change	4
1.5.3 Multiplexing policy	5
1.6 Terminal streams	5
1.6.1 Using the error terminal	5
1.6.2 Terminal class	6
1.7 Directory	6
1.7.1 Reading a directory	6
1.7.2 Creating and removing directories	6
 2 System calls	 9
2.1 Interpreter information	9

2.1.1	Interpreter version	9
2.1.2	Operating system	9
2.1.3	Program information	10
2.2	System calls	10
2.3	Time and date	10
2.3.1	Date representation	10
3	Formatting	13
3.1	Print table object	13
3.1.1	Creating a print table	13
3.1.2	Adding and printing table elements	14
4	Sorting and Searching	15
4.1	Sorting	15
5	Message Digest	17
5.1	Message digest object	17
5.1.1	Creating a message digest	17
5.1.2	Computing a message digest	17
6	Network Services	19
6.1	IP address	19
6.1.1	Domain name system (DNS)	19
6.2	The Address class	19
6.2.1	Name to IP address translation	19
6.2.2	IP address to name translation	20
6.3	Transport layers	20
6.3.1	Service port	21
6.3.2	Host and peer	21
6.4	TCP client socket	21
6.4.1	Day time client	21
6.4.2	HTTP request example	22
6.5	UDP client socket	23
6.5.1	The time client	23
6.5.2	More on reliability	23
6.5.3	Error detection	24
6.6	Socket class	24
6.6.1	Predicates	25
6.7	TCP server socket	25
6.7.1	An echo server	25
6.7.2	The accept method	26
6.7.3	Multiple connections	26
6.8	UDP server socket	26
6.8.1	The echo server	27
6.8.2	Datagram object	27
6.8.3	Input data buffer	27
6.9	Low level socket methods	28

6.9.1	A socket client	28
6.9.2	Other socket methods	29
6.10	Mail delivery	29
6.10.1	A simple mail	29
6.10.2	Recipient address format	30
6.10.3	Message content	30
6.10.4	Message delivery	30
7	Web Services	33
7.1	URL class	33
7.1.1	Character conversion	33
7.1.2	Query string	33
7.2	Generating HTML or XHTML	34
7.2.1	The page header	34
7.2.2	The page body	34
7.2.3	Page generation	34
7.3	Writing CGI scripts	35
7.3.1	Getting the query string	35
7.3.2	Parsing the query string	36
7.3.3	A complete example	36
7.4	Cookie	36
7.4.1	Managing cookies	37
7.4.2	Adding a cookie	37
8	Introduction	39
8.1	Data integration	39
8.2	Basic concepts	39
8.2.1	Cell and data	39
8.2.2	Record	39
8.2.3	Table	39
8.2.4	Collection	40
9	Integration and Importation	41
9.1	Creating a collection	41
9.1.1	The periodic table of elements	41
A	Streams	43
	Input	45
	InputFile	47
	InputMapped	49
	InputString	51
	InputTerm	53
	OutputFile	55
	OutputFile	57
	OutputString	59
	OutputTerm	61
	Terminal	63

Directory	65
Selector	67
B File System Functions	69
dir-p	69
file-p	69
absolute-path	69
relative-path	69
rmfile	70
rmdir	70
C System Classes	71
Time	73
D System Calls	79
exit	79
sleep	79
random	79
get-pid	79
get-env	80
get-host-name	80
get-user-name	80
E Formatting	81
PrintTable	83
Digest	87
F Sorting and searching	89
sort	89
G Networking Classes	91
Address	93
Socket	95
TcpSocket	101
TcpClient	103
TcpServer	105
Datagram	107
UdpSocket	109
UdpClient	111
UdpServer	113
Multicast	115
Mail	117

H	Networking Functions	121
	get-loopback	121
	get-tcp-service	121
	get-udp-service	121
I	WWW/CGI Classes and Functions	123
	Url	125
	CgiQuery	127
	HtmlPage	129
	XHtmlPage	133
	Cookie	135
	Colophon	141

Preface

This manual is part of the *Aleph Programming Language Series*, a multi volume set that describes the programming environment of the **Aleph** system. The entire set contains 4 volumes :

Volume 0 - Aleph Installation Guide is the distribution installation manual.

Volume 1 - Aleph Programmer Guide is the first volume of this set. It is both an introduction and an advanced guide for the the developer.

Volume 2 - Aleph Library Reference is the second volume of this set. It is a complete description of the Aleph standard library.

Volume 3 - Aleph Cross Debugger is the third volume of this set. It is a reference manual to develop and debug Aleph programs.

Volume 4 - Aleph C++ API is the fourth volume of this set. It is a reference manual of the C++ Application Programming Interface (API).

The Aleph programming language

Aleph is a multi-threaded functional programming language with dynamic symbol bindings that support the object oriented paradigm. **Aleph** features a state of the art runtime engine that supports both 32 and 64 bits platforms. **Aleph** comes with a rich set of libraries that are designed to be platform independent. **Aleph** is a free software. A flexible license has been designed for both individuals and corporations. Everybody is encouraged to use, distribute and/or modify the aleph engine for any purpose.

Features

The **Aleph** engine is written in C++ and provides runtime compatibility with it. Such compatibility includes the ability to instantiate C++ classes, use virtual methods and raise or catch exceptions. A comprehensive API has been designed to ease the integration of foreign libraries.

- **Builtin objects**
More than 50 reserved keywords and predicates. Various containers like list, vector, hash table, bitset, and graphs.
- **Functional programming**
Support for *lambda expression* with explicit closure. Symbol scope limitation with *gamma expression*. Form like notation with an easy block declaration.

- **Object oriented**
Single inheritance object mechanism with dynamic symbol resolution. Native class derivation and method override. Static class data member and methods.
- **Multi-threaded engine**
True multi-threaded engine with automatic object protection mechanism against concurrent access. Read and write locking system and thread activation via condition objects.
- **Original regular expression**
Builtin regular expression engine with group matching, exact or partial match and substitution.

Aleph is a core language and libraries. The libraries are a specific set of classes and functions which are structured per application domains. **Aleph** is delivered with a set of standard libraries.

- **aleph-sys**
The `aleph-sys` library is the system calls library. Standard classes and functions are provided to interact with the running machine.
- **aleph-sio**
The `aleph-sio` library is the standard input/output. All input/output operations are performed with this library.
- **aleph-net**
The `aleph-net` library is the networking library. The library is based on the standard *Internet Protocol* and provides various classes to manipulate IP address, client or server sockets.
- **aleph-www**
The `aleph-www` library is the World Wide Web library. The library provides various classes that ease the development of web applications or CGI scripts.
- **aleph-txt**
The `aleph-txt` library is the text processing library. The library provides various functions and classes that ease text manipulation. Sorting data, computing message digest and formatting table is among others, features available in this library.
- **aleph-odb**
The `aleph-odb` library is the object database library. The library provides several objects that can be used to design a database. A client is also provided to directly access the database contents.

Aleph provides *extensions*. An extension is a library or an application which is not installed by default. The user selects during the installation process which extension is needed. For example, the static version of the interpreter is an extension.

Aleph engine

Aleph is an interpreted language. When used interactively, commands are entered on the command line and executed when a complete and valid syntactic object has been constructed. Alternatively, the interpreter can execute a source file. **Aleph** does not have a garbage collector. **Aleph** operates with a lazy, scope based, object destruction mechanism. Each time an object is no longer visible, it is destroyed automatically. At this time, the **Aleph** interpreter is unable to reclaim memory with circular structures. This is a well known problem when using a reference count mechanism. In the future, the **Aleph** engine will provide some mechanisms to resolve this problem.

Flexible Distribution

Aleph is a free software. A flexible license model encourages individuals or corporations to use, copy, modify and/or distribute this software. **Aleph** is designed by software professionals. Quality is one the driving force of the development effort. This is reflected in this distribution by the extensive documentation. A large test suite is used to assess the quality of the distribution. Right now, the engine has been successfully tested on most Linux platforms, Free BSD and Solaris.

License

Aleph is a free software. It can be used, modified and distributed by anybody for personal or commercial use. The only restriction is altering the copyright notice associated with the material. Individual or corporation are permitted to use, include or modify the **Aleph** engine. All material developed with the **Aleph** language belongs to their respective copyright holder.

This program is a free software. it can be redistributed and/or modified, providing that this copyright notice is kept intact. This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose. In no event shall the copyright holder be liable for any direct, indirect, incidental or special damages arising in any way out of the use of this software.

CHAPTER 1

Input/Output Services

This chapter covers the input/output facilities available in the *Standard Input Output* or **aleph-sio** library. The basic operations are related to file manipulations and are later extended to any character input or output streams. Later, various file system calls are described. The **Aleph** i/o library has been designed to be machine independent.

1.1 The aleph-sio library

All Aleph input/output objects are located in the **aleph-sio** library. This library must be loaded prior any operation. Multiple call to the library initialization routine are harmless. The interpreter method `library` loads a specific library by name. When the library has been loaded, the object are available in the **aleph:sio** nameset.

```
interp:library "aleph-sio"
```

1.2 Input and output stream

The **aleph-sio** library is based on facilities provided by two base classes, namely, the Input stream and the Output stream. Both classes have associated predicates with the name `input-p` and `output-p`.

1.2.1 Input stream

The Input base class has several method for reading and testing for character availability. Moreover, the class provides a pushback buffer. Reading character is in the form of three method. `read` without argument returns the next available character or `eof`. With an integer argument, `read` returns a `Buffer` with at most the number of requested characters. The `readln` method returns the next available line.

1.2.2 Output stream

The Output base class provides the base methods to write to an output stream. The `write` method takes literal objects which are automatically converted to string representation and then written to the output stream. Note that for the case of a `Buffer` object, it is the buffer itself that take a stream argument and not the opposite.

1.3 File streams

The `aleph-sio` library provides two classes for file access. The `InputFile` class open a file for input. The `OutputFile` class opens a file for output. The `InputFile` class is derived from the `Input` base class. The **`OutputFile`** class is derived from the `Output` class. By default an output file is created if it does not exist. If the file already exist, the file is truncated to 0. Another constructor for the output file gives more control about this behavior. It takes two boolean flags that defines the truncate and append mode.

```
# load the library
interp:library "aleph-sio"
# create an input file by name
const if (aleph:sio:InputFile "orig.txt")
# create an output file by name
const of (aleph:sio:OutputFile "copy.txt")
```

1.3.1 Stream information

Both **`InputFile`** and **`OutputFile`** supports the `get-name` method which returns the file name.

```
println (if:get-name)
println (of:get-name)
```

Predicates are also available for these classes. The **`input-file-p`** returns true for an input file object. The **`output-file-p`** returns true for an output file object.

```
aleph:sio:input-p      if
aleph:sio:output-p     of
aleph:sio:input-file-p if
aleph:sio:output-file-p of
```

1.3.2 Reading and writing

The `read` method reads a character on an input stream. The `write` method writes one or more literal arguments on the output stream. The `writeln` method writes one or more literal arguments followed by a newline character on the output stream. The `newline` method write a newline character on the output stream. The `eof-p` predicate returns true for an input stream, if the stream is at the end. The `valid-p` predicate returns true if an input stream is in a valid state. With these methods, copying a file is a simple operation.

```
# load the library and open the files
interp:library "aleph-sio"
const if (aleph:sio:InputFile "orig.txt")
const of (aleph:sio:OutputFile "copy.txt")

# loop in the input file and write
while (if:valid-p) (of:write (if:read))
```

The use of the `readln` method can be more effective. The example below is a simple cat program which take the file name an argument.

```
# cat a file on the output terminal
# usage: aleph 0601.als file
```



```
# get the io library
interp:library "aleph-sio"

# cat a file
const cat (name) {
  const f (aleph:sio:InputFile name)
  while (f:valid-p) (println (f:readln))
  f:close
}

# get the file
if (== 0 (interp:argv:length)) {
  errorln "usage: aleph 0601.als file"
} {
  cat (interp:argv:get 0)
}
```

1.4 Input stream status

The input stream provides a general mechanism to test and read for characters. The base method is the `valid-p` predicate that returns `true` if a character can be read from the stream. It is important to understand its behavior which depends on the stream type.

1.4.1 The `valid-p` predicate

Without argument, the `valid-p` predicate checks for an available character from the input stream. This predicate will block if no character is available. On the other end, for a bounded stream like an input file, the method will not block at the end of file. With one integer argument, the `valid-p` predicate will timeout after the specified time specified in milliseconds. This second behavior is particularly useful with unbound stream like socket stream.

1.4.2 The `eof-p` predicate

The `eof-p` predicate does not take argument. The predicate behaves like `not (valid-p 0)`. However, there are more subtle behaviors. For an input file, the predicate will return `true` if and only if a character cannot be read. If a character has been pushed back and the *end-of-file* marker is reached, the method will return `false`. For an input terminal, the method returns `true` if the user and entered the *end-of-file* character (that is Ctrl-D). Once again, the method reacts to the contents of the push-back buffer. For certain input stream, like a TCP socket, the method will return `true` when no character can be read, that is here, the connection has been closed. For an UDP socket, the method will return `true` when all datagram characters have be read.

1.4.3 The `read` method

The `read` method is sometimes disturbing. Nevertheless, the method is a blocking one and will return a character when completed. The noticeable exception is the returned character when an *end-of-file* marker has been reached. The method returns the *Ctrl-D* character. Since a binary file might contains valid character like *Ctrl-D* it is necessary to use the `valid-p` or `eof-p` predicate to check for a file reading completion. This remark apply also to bounded streams like a TCP socket. For

some type of streams like a UDP socket, the method will block when all datagram characters have been consumed and no more datagram has arrived. With this kind of stream, there is no *end-of-file* condition and therefore care should be taken to properly assert the stream content. This last remark is especially true for the `readln` method. The method will return when the *end-of-file* marker is reached, even if a newline character has not been read. With an UDP socket, such behavior will not happen.

1.4.4 Buffer read mode

The `read` method with an integer argument, returns a buffer with at least the number of characters specified as an argument. This method is particularly useful when the contents has a precise size. The method returns a `Buffer` object which can later be used to read, or transform characters. Multi-byte conversion to number should use such approach. The `read` method does not necessarily returns the number of requested characters. Once the buffer is returned, the `length` method can be used to check the buffer size. Note also the existence of the `to-string` method which returns a string representation of the buffer.

```
# try to read 256 characters
const buf (is:read 256)
# get the buffer size
println (buf:length)
# get a string representation
println (buf:to-string)
```

1.5 Multiplexing

I/O multiplexing is the ability to manipulate several streams at the same time and process one at a time. Although the use of threads reduce the needs for i/o multiplexing, there is still situations where they are needed. In other words, I/O multiplexing is identical to the `valid-p` predicate, except that it works with several stream objects.

1.5.1 Selector object

I/O multiplexing is accomplished with the **Selector** class. The constructor takes 0 or several stream arguments. The class manages automatically to differentiate between **Input** stream and **Output** streams. Once the class is constructed, it is possible to get the first stream ready for reading or writing or all of them. We assume in the following example that `is` and `os` are respectively an input and an output stream.

```
# create a selector
const slt (aleph:sio:Selector is)

# at this stage the selector has one stream
# the add method can add more streams
slt:add os
```

The `add` method adds a new stream to the selector. The stream must be either an **Input** or **Output** stream or an exception is raised. The `input-length` method returns the number of input streams in this selector. The `output-length` method returns the number of output streams in this selector. The `input-get` method returns the selector input stream by index. The `output-get` method returns the selector output stream by index.

1.5.2 Waiting for change

The `wait` and `wait-all` methods can be used to detect a status change in the selector. Without argument both methods will block indefinitely until one stream change. With one integer argument, both method blocks until one stream change or the integer argument timeout expires. The timeout is expressed in milliseconds. Note that 0 indicates an immediate return. The `wait` method returns the first stream which is ready either for reading or writing depending whether it is an input or output stream. The `wait-all` method returns a vector with all streams that have changed their status. The `wait` method returns `nil` if the no stream have changed. Similarly, the `wait-all` method returns an empty vector.

```
# wait for a status change
const is (slt:wait)
# is is ready for reading - make sure it is an input one
if (aleph:sio:input-p is) (is:read)
```

A call to the `wait` method will always returns the first input stream (if any).

1.5.3 Multiplexing policy

When used with several input streams in a multi-threaded context, the selector behavior can becomes quite complicated. Either `wait` and `wait-all` methods check first the input streams push-back buffer. If one or several buffer is (are) not empty, the method returns with these streams. During this operation, the input streams are locked, so no other thread can push-back a character. The selector then checks for status change and unlock the streams. Note that the output streams are not locked. Note also that a thread which rely on the input stream push-back method to release a selector will result in a dead lock.

1.6 Terminal streams

Terminal streams are another kind of streams available in the standard input/output library. The **InputTerm**, **OutputTerm** and **ErrorTerm** classes are low level classes used to read or write from or to the standard streams. The basic methods to read or write are the same as the file streams. Reading from the input terminal is not a good idea, since the class does not provide any formatting capability. One may prefer to use the **Terminal** class. The use of the output terminal or error terminal streams is convenient when the interpreter standard streams have been changed but one still need to print to the terminal.

1.6.1 Using the error terminal

The **ErrorTerm** class is the most frequently used class for printing data on the standard error stream. Aleph provides the reserved keyword **error** or **errorln** to write on the interpreter error stream. If the interpreter error stream has been changed, the use of the **ErrorTerm** will provide the facility required to print directly on the terminal. The `cat` program can be rewritten to do exactly this.

```
# cat a file on the error terminal

# get the io library
interp:library "aleph-sio"

# cat a file
const cat (name es) {
```

```

const f (aleph:sio:InputFile name)
while (f:valid-p) (es:writeln (f:readln))
f:close
}

```

1.6.2 Terminal class

The **Terminal** class combines an input stream and an output stream with some line editing capabilities. When the class is created, the constructed attempts to detect if the input and output streams are bounded to a terminal (i.e tty). If the line editing capabilities can be loaded (i.e non canonical mode), the terminal is initialized for line editing. Arrows, backspace, delete and other control sequences are available when using the `readline` method. The standard methods like `read` or `readln` do not use the line editing features. When using a terminal, the prompt can be set to whatever the user wishes with the methods `set-primary` or `set-secondary`. A secondary prompt is displayed when the `readline` method is called with the boolean argument `false`.

```

const term (Terminal)
term:set-primary "demo:"
const line (term:readline) → demo:
errorln line

```

1.7 Directory

The **Directory** class provides a facility to manipulate directories. A directory object is created either by name or without argument by considering the current working directory. Once the directory object is created, it is possible to retrieve its contents, create new directory or remove empty one.

1.7.1 Reading a directory

A **Directory** object is created either by name or without argument. With no argument, the current directory is opened. This is the best method compared to `"."`. When the current directory is opened, its full name is computed internally and can be retrieved with the `get-name` method.

```

# print the current directory
const pwd (aleph:sio:Directory)
println (pwd:get-name)

```

Once the directory object is opened, it is possible to list its contents. The `get-list` method returns the full contents of the directory object. The `get-files` method returns a list of files in this directory. The `get-subdirs` method returns a list of sub directories in this directory.

```

# print a list of files
const pwd (aleph:sio:Directory)
const lsf (d:get-files)
for (name) (lsf) (println name)

```

1.7.2 Creating and removing directories

The `mkdir` and `rmdir` methods can be used to create or remove a directory. Both methods take a string argument and construct a full path name from the directory name and the argument. This

approach has the advantage of being file system independent. If the directory already exists, the `mkdir` methods succeeds. The `rmdir` method requires the directory to be empty.

```
const tmp (aleph:sio:Directory (aleph:sio:absolute-path "tmp"))
const exp (tmp:mkdir "examples")
const lsf (exp:get-files)
println   (lsf:length) → 0
tmp:rmdir "examples"
```

The function `absolute-path` constructs an absolute path name from the argument list. If relative path needs to be constructed, the function `relative-path` might be used instead.

CHAPTER 2

System calls

This chapter covers the system facilities available in the **aleph-sys** library. The basic operations that are embedded in the interpreter gives system information. Complex information, like the system time are provided via specific classes.

2.1 Interpreter information

The **Aleph** interpreter provides a set reserved names that are related to the system platform. Example `0501.als` demonstrates the available information.

```
zsh > aleph 0501.als
major version number      : 0
minor version number      : 9
patch version number      : 0
interpreter version        : 0-9-0
program name              : aleph
operating system name     : linux
operating system type     : unix
aleph official url        : http://www.aleph-lang.org
```

2.1.1 Interpreter version

The interpreter version is identified by 3 numbers called *major*, *minor* and *patch* numbers. A change in the major number represents a major change in the **Aleph** language. The minor number indicates a major change in the interface or libraries. A change in the patch number indicates bug fixes. All values are accessed via the interpreter itself. The `major-version`, `minor-version`, `patch-version` symbols are bound to these values.

```
println "major version number      : " interp:major-version
println "minor version number      : " interp:minor-version
println "patch version number      : " interp:patch-version
```

2.1.2 Operating system

The operating system is uniquely identified by its name. The operating system type (or category) uniquely identifies the operating system flavor. At this time, only UNIX like system are supported. The operating system name can be either `linux`, `freebsd` or `solaris`.

```
println "operating system name : " interp:os-name
println "operating system type : " interp:os-type
```

2.1.3 Program information

Program information are carried by two symbols that identifies the program name and the official **Aleph** URL. While the first might be useful, the second one is mostly used by demo programs.

```
println "program name          : " interp:program-name
println "aleph official url    : " interp:aleph-url
```

2.2 System calls

The **aleph-sys** library provides various system calls that cannot be classified into any particular category.

Table 1 Aleph system call functions

Function	Description
exit	exit unconditionally with an exit code
sleep	pause for a certain time
random	return a random integer number
get-pid	get the process identifier
get-env	get an environment variable
get-host-name	return the host name
get-user-name	return the user name

2.3 Time and date

The **Time** class is special class that represent the system date in utc or local format. Numerous methods are provided to access a particular field, like hour, minute, day in month, week etc. Without argument the time instance is constructed with the current system time. An integer argument can be used to force a particular time.

2.3.1 Date representation

Once a time instance is constructed, various formats methods can returns the date, time or both. The *format-date* and *format-time* returns a formatted string for the local date and time. The *utc-format-date* and *utc-format-time* do the same in UTC.

```
aleph >interp:library "aleph-sys"
aleph >const time (aleph:sys:Time)
aleph >println (time:format-date)
6/5/2003
aleph >println (time:format-time)
22:11:30
```

Another form of date representation is the one specified by RFC 822. That format contains both the time and date. Note that the date is formatted in UTC (improperly called GMT).


```
aleph >interp:library "aleph-sys"  
aleph >const time (aleph:sys:Time)  
aleph >println (time:utc-format-rfc)  
Wed, 06 Jun 2003 05:11:30 GMT
```

Other methods are available to query the date and time information. These are described in the reference manual.

CHAPTER 3

Formatting

This chapter is dedicated to the **Aleph** text and data formatting, a subpart of the *text processing* library. The first part of this chapter covers the *print table* object.

3.1 Print table object

The `PrintTable` class is a formatting class for tables. The table is constructed with the number of columns (default to 1) and eventually the number of rows. Once the table is created, element are added to the table with the `add` method. Specific table element can be set with the `set` method. The class provide a `format` method those default is to print the table on the interpreter standard output. With an output stream argument or a buffer, the table is formatted to these objects. The table formatting includes an optional column width, a filling character and a filling direction flag. By default, the column width is 0. This means that the column width is computed as the maximum length of all column elements. If the column width is set with the `set-column-size` method, the string element might be truncated to the left or right (depending on the filling flag) to fit the column width.

3.1.1 Creating a print table

The table is created with 0, one or two arguments. Without argument, the table has one column. The first argument is the number of columns. The optional second argument is the desired number of rows.

```
# a one column table
const tbl-1 (aleph:txt:PrintTable)
# a five columns table
const tbl-5 (aleph:txt:PrintTable 5)
# a five columns x 3 rows table
const tbl-5x3 (aleph:txt:PrintTable 5 3)
```

Once the table is created, the column size can be set. For example, if the previous 5 columns table must have the first column with 10 characters, the `set-column-size` method can be used to do so. Additionnaly, the `set-column-direction` can also be used to indicate a right filling. By default, filling characters are placed on the left of the string, therefore producing a right alignment.

```
# reset column 0 to a size 10
tbl-5:set-column-size 0 10
# set left alignment, aka right filling
```

```
tbl-5:set-column-direction 0 true
```

3.1.2 Adding and printing table elements

The add method is used to add literals to the table. Without argument, a new row is created and the row index is returned. With one or several literal, a new row is created and the arguments inserted into the table. The number of arguments must match the number of columns. The next example shows a simple flight time table (my preferred destinations).

```
# load the text processing library
interp:library "aleph-txt"

# create a new print table with 3 columns
const tbl (aleph:txt:PrintTable 3)

# add the rows
tbl:add "Planet" "Diameter" "Rotation time"
tbl:add "Mercury" 4840 "1407:36"
tbl:add "Venus" 12400 "5819:51"
tbl:add "Earth" 12756 "23:56"
tbl:add "Mars" 6800 "24:37"
tbl:add "Jupiter" 142800 "9:50"
tbl:add "Saturn" 120800 "10:14"
tbl:add "Uranus" 47600 "10:49"
tbl:add "Neptune" 44600 "15:40"
tbl:add "Pluto" 5850 "153:17"

# set the table format
tbl:set-column-size 0 10
tbl:set-column-size 1 10
tbl:set-column-direction 2 true

# print the table
tbl:format
```

The format method prints the formatted table. Without argument, the interpreter standard output is used.

```
zsh> aleph txt-0001.als
Planet      Diameter      Rotation time
Mercury     4840                1407:36
Venus       12400               5819:51
Earth       12756               23:56
Mars        6800                24:37
Jupiter     142800              9:50
Saturn      120800              10:14
Uranus      47600               10:49
Neptune     44600               15:40
Pluto       5850               153:17
```

Note how the columns are formatted. Column 2 has the flag set to true while the others have the default flag set to false.

CHAPTER 4

Sorting and Searching

This chapter is dedicated to the **Aleph** sorting and searching engine, a subpart of the *text processing* library. All objects and functions are part of the `aleph:txt` nameset.

4.1 Sorting

The **sort** function operates with a vector object and sorts the elements in ascending order. Any kind of objects can be sorted as long as they support a comparison method. The elements are sorted in place by using a `quick sort` algorithm.

```
# create an unsorted vector
const v-i (Vector 7 5 3 4 1 8 0 9 2 6)
# sort the vector in place
aleph:txt:sort v-i
# print the vector
for (e) (v) (println e)
```

CHAPTER 5

Message Digest

This chapter is dedicated to the **Aleph** message digest computation, a subpart of the *text processing* library. The first part of this chapter covers the *Digest* object.

5.1 Message digest object

The `Digest` class is a message digest computation class. By default, the MD5 algorithm as defined by RFC 1321 is bound to the class. The message digest class computes a *message digest* from an input string or a buffer. The message digest is returned as a string.

5.1.1 Creating a message digest

By default a message digest is created with support for the MD5 algorithm. No argument is passed to the constructor.

```
# get a default digest (MD5)
const md (aleph:txt:Digest)
```

5.1.2 Computing a message digest

The `compute` method computes a message digest from an input string or a buffer. For example, the string "hello world" returns the message digest "5EB63BBBE01EEED093CB22BB8F5ACDC3"

```
const digest (md:compute "hello world")
```

CHAPTER 6

Network Services

This chapter is dedicated to the **Aleph** networking services. It assumes that the reader has a basic knowledge of the *Internet Protocol or (IP)*. The **Aleph** implementation provides, in a single library called `aleph-net`, all classes and functions needed to perform IP operations, create server or clients programs. This library is also designed to support *IPv6* with certain platforms (Currently Linux 2.2, FreeBSD 4.x and Solaris 5.8).

6.1 IP address

The IP based communication uses a standard address to reference a particular peer. With IP version 4 *IPv4*, the standard dot notation is with 4 bytes. With IP version 6 *IPv6*, the standard semicolon notation is with 16 bytes. The current **Aleph** implementation supports both versions. Even if your platform supports IPv6, it does not mean that it is enabled. You should consult your *system administration* guide to do so. Generally, this involves setting the `/etc/hosts` file and activating the `inet6` option in the `/etc/resolv.conf`.

```
127.0.0.1      → ipv4 localhost
0:0:0:0:0:0:0:1 → ipv6 localhost
```

IP address architecture and behavior are described in various documents as listed in the bibliography.

6.1.1 Domain name system (DNS)

The translation between a host name and an IP address is performed by a *resolver* which uses the *Domain Name System or (DNS)*. Access to the DNS is automatic with the **Aleph** implementation. Depending on the machine resolver configuration, a particular domain name translation might result in an IPv4 or IPv6 address. As of today, the user might expect to get only IPv4 address (UNIX system requires the `resolv.conf` file to have the `inet6` option active to get an IPv6 address. Using this option can trigger some unexpected behavior).

The mapping between an IP address and a host name returns the associated *canonical name* for that IP address. This is the reverse of the preceding operation.

6.2 The Address class

The `aleph-net:Address` class allows manipulation of IP address. The constructor takes a string as its arguments. The argument string can be either an IP address or a host name (qualified or not). When the address is constructed with a host name, the IP address resolution is done immediately.

6.2.1 Name to IP address translation

The most common operation is to translate a host name to its equivalent IP address. Once the Address object is constructed, the `get-ip-address` method returns a string representation of the internal IP address. The following example prints the IP address of the localhost, that is 127.0.0.1 with IPv4.

```
# load network library
interp:library "aleph-net"

# get the localhost address
const addr (aleph:net:Address "localhost")

# print the ip address
println (addr:get-ip-address)
```

As another example, the `aleph:sys:get-host-name` function returns the host name of the running machine. The previous example can be used to query its IP address.

6.2.2 IP address to name translation

The reverse operation of name translation maps an IP address to a *canonical name*. The `get-canonical-name` method of the Address class returns such name. Example 3101.als is a demonstration program which prints the address original name, the IP address and the canonical name. Feel free to use it with your favorite site to check the equivalence between the original name and the canonical name.

```
# print the ip address information of the arguments
# usage: aleph 3101.als [hosts ...]

# get the network library
interp:library "aleph-net"

# print the ip address
const ip-address-info (host) {
  try {
    const addr (aleph:net:Address host)
    println "host name      : " (addr:get-name)
    println "  ip address   : " (addr:get-ip-address)
    println "  canonical name : " (addr:get-canonical-name)
  } (errorln "error: " what:reason)
}

# get the hosts
for (s) (interp:argv) (ip-address-info s)

zsh> aleph 3101.als localhost www.aleph-lang.org
host name      : localhost
ip address     : 127.0.0.1
canonical name : localhost
host name      : www.aleph-lang.org
ip address     : 216.15.47.53
canonical name : www.aleph-lang.org
```

6.3 Transport layers

The two transport layer protocols supported by the *Internet protocol* is the **TCP**, a full-duplex oriented protocol, and **UDP**, a datagram protocol. TCP is a reliable protocol while UDP is not. By reliable, we mean that the protocol provides automatically some mechanisms for error recovery, message delivery, acknowledgment of reception, etc... The use of TCP vs. UDP is dictated mostly by the reliability concerns, while UDP reduces the traffic congestion.

6.3.1 Service port

In the client-server model, a connection is established between two hosts. The connections is made via the *IP address* and the *port number*. For a given service, a port identifies that service at a particular address. This means that multiple services can exist at the same address. More precisely, the transport layer protocol is also used to distinguish a particular service.

The **Aleph** network library provides a simple mechanism to retrieve the port number, given its name and protocol. The function `get-tcp-service` and `get-udp-service` returns the port number for a given service by name. For example, the `daytime` server is located at port number 13.

```
assert 13 (aleph:net:get-tcp-service "daytime")
assert 13 (aleph:net:get-udp-service "daytime")
```

6.3.2 Host and peer

With the client server model, the only information needed to identify a particular client or server is the address and the port number. When a client connects to a server, it specify the port number the server is operating. The client uses a random port number for itself. When a server is created, the port number is used to bind the server to that particular port. If the port is already in use, that binding will fail. From a reporting point of view, a connection is therefore identified by the running host address and port, and the peer address and port. For a client, the peer is the server. For a server, the peer is the client.

6.4 TCP client socket

The `TcpClient` class creates an TCP client object by address and port. The address can be either a string or an `Address` object. During the object construction, the connection is established with the server. Once the connection is established, the client can use the `read` and `write` method to communicate with the server. The `TcpClient` class is derived from the `Socket` class which is derived from the `Input` and `Output` classes.

6.4.1 Day time client

The simplest example is a client socket which communicates with the *daytime* server. The server is normally running on all machines and is located at port 13.

```
# get the network library
interp:library "aleph-net"

# get the daytime server port
const port (aleph:net:get-tcp-service "daytime")
```

```
# create a tcp client socket
const s (aleph:net:TcpClient "localhost" port)

# read the data - the server close the connection
while (s:valid-p) (println (s:readln))
```

Example 3201.als in the example directory prints the day time of the local host without argument or the day time of the argument. Feel free to use it with www.aleph-lang.org. If the server you are trying to contact does not have a day time server, an exception will be raised and the program terminates.

```
zsh> aleph 3201.als www.aleph-lang.org
```

6.4.2 HTTP request example

Another example which illustrates the use of the `TcpClient` object is a simple client which download a web page. At this stage we are not concern with the URL but rather the mechanics involved. The request is made by opening a TCP client socket on port 80 (the HTTP server port) and sending a request by writing some HTTP commands. When the commands have been sent, the data sent by the server are read and printed on the standard output. Note that this example is not concerned by error detection.

```
# fetch an html page by host and page
# usage: aleph 3203.als [host] [page]

# get the network library
interp:library "aleph-net"
interp:library "aleph-sys"

# connect to the http server and issue a request
const send-http-request (host page) {
  # create a client sock on port 80
  const s      (aleph:net:TcpClient host 80)
  const saddr (s:get-socket-address)

  # format the request
  s:writeln "GET " page " HTTP/1.1"
  s:writeln "Host: " (saddr:get-canonical-name)
  s:writeln "Connection: close"
  s:writeln "User-Agent: aleph tcp client example"
  s:newline

  # write the result
  while (s:valid-p) (println (s:readln))
}

# get the argument
if (!= (interp:argv:length) 2) (aleph:sys:exit 1)
const host (interp:argv:get 0)
const page (interp:argv:get 1)

# send request
send-http-request host page
```

6.5 UDP client socket

UDP client socket is similar to TCP client socket. However, due to the unreliable nature of UDP, UDP clients are somehow more difficult to manage. Since there is no flow control, it becomes more difficult to assess whether or not a datagram has reached its destination. The same apply for a server, where a reply datagram might be lost. The `UdpClient` class is the class which creates a UDP client object. Its usage is similar to the `TcpClient`.

6.5.1 The time client

The UDP time server normally runs on port 37 is the best place to enable it. A UDP client is created with the `UdpClient` class. Once the object is created, the client sends an empty datagram to the server. The server send a reply datagram with 4 bytes, in network byte order, corresponding to the date as of January 1st 1900. Example `3204.als` prints date information after contacting the local host time server or the host specified as the first argument.

```
# print the time with a udp client socket

# get the libraries
interp:library "aleph-net"
interp:library "aleph-sys"

# get the daytime server port
const port (aleph:net:get-udp-service "time")

# create a client socket and read the data
const print-time (host) {
  # create a udp client socket
  const s (aleph:net:UdpClient host port)
  # send an empty datagram
  s:write
  # read the 4 bytes data and adjust to epoch
  const buf (s:read 4)
  const val (- (buf:get-quad) 2208988800)
  # format the date
  const time (aleph:sys:Time val)
  println (time:format-date) ' ' (time:format-time)
}

# check for one argument or use localhost
const host (if (== (interp:argv:length) 0)
               "localhost" (interp:argv:get 0))
print-time host
```

This example calls for several comments. First the `write` method without argument sends an empty datagram. It is the datagram which trigger the server. The `read` method reads 4 bytes from the reply datagram and places them in a `Buffer` object. Since the bytes are in network byte order, the conversion into an integer value is done with the `get-quad` method. Finally, in order to use the `Time` class those epoch is January 1st 1970, the constant 2208988800 is subtracted from the result. Remember that the time server sends the date in reference to January 1st 1900. More information about the time server can be found in *RFC738*.

6.5.2 More on reliability

The previous example has some inherent problems due to the unreliability of UDP. If the first datagram is lost, the `read` method will block indefinitely. Another scenario which causes the `read` method to block is the loss of the server reply datagram. Both problem can generally be fixed by checking the socket with a timeout using the `valid-p` method. With one argument, the method timeout and return false. In this case, a new datagram can be send to the server. Example 3205.als illustrates this point. We print below the extract of code.

```
# create a client socket and read the data
const print-time (host) {
  # create a udp client socket
  const s (aleph:net:UdpClient host port)
  # send an empty datagram until the socket is valid
  s:write
  # retransmit datagram each second
  while (not (s:valid-p 1000)) (s:write)
  # read the 4 bytes data and adjust to epoch
  const buf (s:read 4)
  const val (- (buf:get-quad) 2208988800)
  # format the date
  const time (aleph:sys:Time val)
  println (time:format-date) ' ' (time:format-time)
}
```

Note that this solution is a naive one. In the case of multiple datagrams, a sequence number must be placed because there is no clue about the lost datagram. A simple rule of thumb is to use TCP as soon as reliability is a concern, but this choice might not so easy.

6.5.3 Error detection

Since UDP is not reliable, there is no simple solution to detect when a datagram has been lost. Even worse, if the server is not running, it is not easy to detect that the client datagram has been lost. In such situation, the client might indefinitely send datagram without getting an answer. One solution to this problem is again to count the number of datagram re-transmit and eventually give up after a certain time.

6.6 Socket class

The `Socket` class is the base class for both the `TcpClient` and `UdpClient`. The class provides methods to query the socket port and address as well as the peer port and address. Note at this point that the UDP socket is a connected socket. Therefore, these methods will work fine. The `get-socket-address` and `get-socket-port` returns respectively the address and port of the connected socket. The `get-peer-address` and `get-peer-port` returns respectively the address and port of the connected socket's peer. Example 3206.als illustrates the use of these methods.

```
# create a client socket and read the data
const print-socket-info (host) {
  # create a tcp client socket
  const s (aleph:net:TcpClient host port)
  # print socket address and port
  const saddr (s:get-socket-address)
  const sport (s:get-socket-port)
  println "socket ip address      : " (saddr:get-ip-address)
```

```

println "socket canonical name : " (saddr:get-canonical-name)
println "socket port           : " sport
# print peer address and port
const paddr (s:get-peer-address)
const pport (s:get-peer-port)
println "peer ip address       : " (paddr:get-ip-address)
println "peer canonical name   : " (paddr:get-canonical-name)
println "peer port             : " pport
}

```

6.6.1 Predicates

The `Socket` class is associated with the `socket-p` predicate. The respective client objects have the `tcp-client-p` predicate and `udp-client-p` predicate.

6.7 TCP server socket

The `TcpServer` class creates an TCP server object. There are several constructors for the TCP server. In its simplest form, without port, a TCP server is created on the `localhost` with an ephemeral port number (i.e port 0 during the call). With a port number, the TCP server is created on the `localhost`. For a multi-homed host, the address to use to run the server can be specified as the first argument. The address can be either a string or an `Address` object. In both cases, the port is specified as the second argument. Finally, a third argument called the *backlog* can be specified to set the number of acceptable incoming connection. That is the maximum number of pending connection while processing a connection. The following example shows various ways to create a TCP server.

```

trans s (aleph:net:TcpServer)
trans s (aleph:net:TcpServer 8000)
trans s (aleph:net:TcpServer 8000 5)
trans s (aleph:net:TcpServer "localhost" 8000)
trans s (aleph:net:TcpServer "localhost" 8000 5)
trans s (aleph:net:TcpServer (Address "localhost") 8000)
trans s (aleph:net:TcpServer (Address "localhost") 8000 5)

```

6.7.1 An echo server

A simple *echo server* can be built and tested with the standard *telnet* application. We wish to echo all line we type within the *telnet* client. The server is bound on the port 8000 (Note that port 0 to 1024 are privileged ports and can only be used by `root`). Example `3301.als` is the server example.

```

# get the network library
interp:library "aleph-net"

# create a tcp server on port 8000
const srv (aleph:net:TcpServer 8000)

# wait for a connection
const s (srv:accept)

# echo the line until the end

```

```
while (s:valid-p) (s:writeln (s:readln))
```

The *telnet* session is then quite simple. The line `hello world` is echoed by the server.

```
zsh> telnet localhost 8000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
hello world
^C
zsh>
```

6.7.2 The accept method

The previous example illustrates the mechanics of a server. When the server is created, the server is ready to accept connection. The `accept` method blocks until a client connect with the server. When the connection is established, the `accept` method returns a socket object which can be used to read and write data.

6.7.3 Multiple connections

One problem with the previous example is that the server accepts only one connection. In order to accept multiple connection, the `accept` method must be placed in a loop, and the server operation in a thread (There are some situations where a new process might be more appropriate than a thread). Example 3302.als illustrates such point.

```
# get the network library
interp:library "aleph-net"

# this function echo a line from the client
const echo-server (s) {
  while (s:valid-p) (s:writeln (s:readln))
}

# create a tcp server on port 8000
const srv (aleph:net:TcpServer 8000)

# wait for a connection
while true {
  trans s (srv:accept)
  launch (echo-server s)
}
```

6.8 UDP server socket

The `UdpServer` class is similar to the `TcpServer` object, except that there is no backlog parameters. In its simplest form, the UDP server is created on the `localhost` with an ephemeral port (i.e port 0). With a port number, the server is created on the `localhost`. For a multi-homed host, the address used to run the server can be specified as the first argument. The address can be either a string or an `Address` object. In both cases, the port is specified as the second argument.


```
trans s (aleph:net:UdpServer)
trans s (aleph:net:UdpServer 8000)
trans s (aleph:net:UdpServer "localhost" 8000)
trans s (aleph:net:UdpServer (Address "localhost") 8000)
```

6.8.1 The echo server

The *echo server* can be revisited to work with udp datagram. The only difference is the use of the `accept` method. For a UDP server, the method return a datagram object which can be used to read and write data.

```
# get the network library
interp:library "aleph-net"

# create a udp server on port 8000
const srv (aleph:net:UdpServer 8000)

# wait for a connection
while true {
  trans dg (srv:accept)
  dg:writeln (dg:readln)
}
```

6.8.2 Datagram object

With a UDP server, the `accept` method returns a *Datagram object*. Because a UDP is connection-less, the server has no idea from whom the datagram is coming until that one has been received. When a datagram arrives, the *Datagram* object is constructed with the peer address being the source address. Standard i/o methods can be used to read or write. When a write method is used, the data are sent back to the peer in a form of another datagram.

```
# wait for a datagram
trans dg (s:accept)

# assert datagram type
assert true (datagram-p dg)

# get contents length
println "datagram buffer size : " (dg:get-buffer-length)

# read a line from this datagram
trans line (dg:readln)

# send it back to the sender
s:writeln line
```

The following table summarize the datagram methods.

6.8.3 Input data buffer

For a datagram, and generally speaking, for a UDP socket, all input operations are buffered. This means that when a datagram is received, the *accept* method places all data in an input buffer. This

Method	Description
<code>read</code>	read one character
<code>read <i>size</i></code>	read <i>n</i> characters and return a buffer
<code>readln</code>	read a line
<code>write</code>	send an empty datagram
<code>write <i>Literal</i></code>	write one or more literal objects
<code>writeln</code>	same as <code>write</code> plus a newline character
<code>newline</code>	write a newline character
<code>valid-p</code>	return true if some data are available
<code>eof-p</code>	return true if no data is available
<code>get-buffer-length</code>	return the input buffer length
<code>pushback</code>	pushback one character or a string

means that a read operation does not necessarily flush the whole buffer but rather consumes only the requested character. For example, if one datagram contains the string *hello world*. A call to `readln` will return the entire string. A call to `read` will return only the character 'h'. Subsequent call will return the next available characters. A call like `read 5` will return a buffer with 5 characters (i.e the string *hello*). Subsequent call will return the remaining string. In any case, the `get-buffer-length` will return the number of available characters in the buffer. A call to `valid-p` will return true if there are some characters in the buffer or if a new datagram has arrived.

Care should be taken with the `read` method. For example if there is only 4 characters in the input buffer and a call to `read` for 10 characters is made, the method will block until a new datagram is received which can fill the remaining 6 characters. Such situation can be avoided by using the `get-buffer-length` and the `valid-p` methods. Remember also that a timeout can be specified with the `valid-p` method.

6.9 Low level socket methods

Some folks always prefer to do everything by themselves. Most of the time for good reasons. If this is your case, you might have to use the low level socket methods. Instead of using a client or server class, the **Aleph** implementation let's you create a **TcpSocket** or **UdpSocket**. Once this done, the `bind`, `connect` and other methods can be used to create the desired connection.

6.9.1 A socket client

A simple TCP socket client is created with the **TcpSocket** class. Then the `connect` method is called to establish the connection.

```
# create an address and a tcp socket
const addr (aleph:net:Address "localhost")
const sid (aleph:net:TcpSocket)
# connect the socket
sid:connect 13 addr
```

Once the socket is connected, normal read and write operations can be performed. After the socket is created, it is possible to set some options. A typical one is `NO-DELAY` which disable the Naggle algorithm.

```
# create an address and a tcp socket
const addr (aleph:net:Address "localhost")
```

```

const sid (aleph:net:TcpSocket)
# disable the naggle algorithm
sid:set-option sid:NO-DELAY true
# connect the socket
sid:connect 13 addr

```

6.9.2 Other socket methods

Other socket methods are available. The `bind` and `listen` methods can be used to create a server. The table below is a resume of the socket methods.

Method	Description
<code>bind</code>	bind this socket
<code>connect</code>	connect this socket
<code>ipv6-p</code>	check for ipv6 socket
<code>read</code>	returns the next available character
<code>readln</code>	returns the next available line
<code>write</code>	write a character or a string
<code>writeln</code>	write a string followed by a newline
<code>newline</code>	write a new line character
<code>close</code>	close this socket
<code>valid-p</code>	returns true if a character is available
<code>eof-p</code>	returns true if the socket has been closed
<code>pushback</code>	pushback a character or a string
<code>shutdown</code>	shutdown a connection
<code>get-buffer-length</code>	return the read buffer length
<code>get-socket-address</code>	return the socket address
<code>get-socket-port</code>	return the socket port
<code>get-peer-address</code>	return the peer address
<code>get-peer-port</code>	return the peer port
<code>set-option</code>	set a socket option

6.10 Mail delivery

The `Mail` class is a mail delivery object which manages to contact an MTA *Mail Transport Agent* in order to deliver a message to one or several recipients. By default, the object contacts the local MTA, but this behavior can be changed with the `set-mta-address` method. The class implements the recipient address syntax as specified by RFC822.

6.10.1 A simple mail

At construction, the instance is empty. Only the recipient address needs to be specified. The `send` method send the message by contacting the MTA. If an error occurs, an exception is raised.

```

# get the network library
interp:library "aleph-net"

# create an empty mail
const mail (aleph:met:Mail)

```

```
# add the recipient address
mail:to "me@domain.org"
```

```
# send the message
mail:send
```

An empty message is sent to *me@domain.org*. By default, the subject is initialized to "no subject".

6.10.2 Recipient address format

RFC822 defines the recipient address format. The simplest one is a local user or a qualified name with a domain. The **Mail** object takes care of detecting the presence of the < and > characters. If a string precedes the address, the enclosed address is used to communicate with the MTA, but the original one is placed in the header. The following example illustrates various address format.

```
mail:to "me"
mail:to "<me>"
mail:to "me@domain.org"
mail:to "<me@domain.org>"
mail:to "user <me@domain.org>, other <other@domain.org>"
```

The `to` method adds an address to the direct recipient list. Several call to this method or several address in one call can be made. In the case of multiple addresses in one call, a comma ',' is used as the address separator. The `cc` method adds one or several addresses to the recipients copy list. This list is also added in the header. The `bcc` method adds one or several addresses to the recipient *blind* copy list. This list is not included in the header.

6.10.3 Message content

The message is built by specifying the subject and filling the message buffer. The `subject` method take a string argument to be used as the message subject. The `add` and `addln` methods add one or several literals to the message buffer. The `addln` method adds a new-line character at the end. Because literals are used with this method, multiple arguments can be used as well as native representation. This method behaves like the `write` method of an output stream.

```
# set message subject
mail:subject "a simple mail demo"
# add a line in the message buffer
mail:add "This line is a text added to the message"
mail:addln "a simple number: " 123 "is automatically converted"
```

6.10.4 Message delivery

The `send` method contacts the MTA and request a message delivery. Example 3303.als illustrates a complete use of the **Mail** class.

```
# send an email to yourself

# get the libraries
interp:library "aleph-sys"
interp:library "aleph-net"
```

```
# get your user name
const user-name (aleph:sys:get-user-name)

# prepare the mail
const mail (aleph:net:Mail)
mail:to      user-name
mail:subject "hello from aleph example"
mail:addln   "This is a generated message from the Aleph"
mail:addln   "mail object - Enjoy the ride"
mail:addln   "The Aleph team"

# send the mail
mail:send
```

CHAPTER 7

Web Services

This chapter covers the **Aleph** Web services. Web services are designed to handle *CGI* scripts. Since a CGI script works with the standard input and output stream, there is no particular device operations described in this chapter. We assume that the reader has a basic knowledge of CGI operations. All objects described in this chapter belong to the **aleph-www** library. The **aleph:www** nameset is used to bind this library.

7.1 URL class

The **URL** class is a special class that parses a *Uniform Resource Locator or URL* string and provides methods to access individual components of that URL. The URL object is constructed with the string to parse.

```
const url (aleph:www:Url "http://www.aleph-lang.org")
```

An URL can be broken into several components called the *scheme*, the *host*, optionally the *port*, the *path*, optionally the *query* and the *fragment*. The URL class provides a method to retrieve each component of the parsed URL.

```
const url (aleph:www:Url "http://www.aleph-lang.org")
println (url:get-scheme) → http
println (url:get-host)   → www.aleph-lang.org
println (url:get-port)   → 80
println (url:get-path)   → /
```

Note that in the previous example, the port and path are not specified. If the scheme is `http`, the default values of 80 and `/` are returned.

7.1.1 Character conversion

The URL class performs automatically the character conversion in the input URL. For example, the `'+'` character is replaced by a blank. The `'%'` character followed by two hexadecimal values is replaced by the corresponding ASCII character. Note that this conversion does not apply to the query string.

7.1.2 Query string

The `get-query` method returns the query string of the URL. The query string starts after the `'?'` character. The query string is a series of key-pair values separated by the `'&'` character.

```
const url (aleph:www:Url
    "http://www.aleph-lang.org?name=hello&value=world")
println (url:get-query) → name=hello&value=world
```

The Web service library also provides a query string class which parse a query string.

7.2 Generating HTML or XHTML

The **HtmlPage** class is the primary interface to generate *HTML* code. The class operates by filling the header and the body of the page with HTML statements. Several methods are provided to ease the task of the page generation. The HTML version is assumed to be *strict 4.01*. Since HTML 4.01 is designed to work with *style sheet*, the user must be prepared to handle this when generating its own HTML page. A derived class name **XHtmlPage** can be used to produce XHTML 1.0 page. The interface is the same as the **HtmlPage** class.

7.2.1 The page header

The **HtmlPage** or **XHtmlPage** constructor takes no argument. The basic method used to add something in the header is the `add-head` method which take one or several literal arguments. The `add-title` method adds a title to the header. The `add-style` adds the style sheet definition to the header. The `add-author` add the author's name to the page header. Finally, the `add-meta` method adds a *meta* statement to the header in the form of name and content.

```
# create a new html page
const page (aleph:www:HtmlPage)
# add the title
page:add-title "An HTML page example"
# add the author
page:add-author "The doc author"
# add the style sheet
page:add-style "style.css"
```

7.2.2 The page body

The only method provided to access the page body is the `add-body` method. The method takes one or several literal arguments. These arguments are used to fill the page body.

```
# add a simple message to the page
page:add-body "<p class='title'>Hello World"
```

7.2.3 Page generation

The `write-page` method write the complete HTML page. The `write-head` only writes the header and the `write-body` only writes the page body. The `get-buffer` method returns the page in a buffer object. A special method called `write-cgi` can be used inside a CGI script to write the HTML page. The following example is a resume of the previous examples.

```
# get the library
```



```

interp:library "aleph-www"
# create a new html page
const page (aleph:www:HtmlPage)
# add the title
page:add-title "An HTML page example"
# add the author
page:add-author "The doc author"
# add the style sheet
page:add-style "/style.css"
# add a simple message to the page
page:add-body "<p class='title'>Hello World"
# write the page
page:write-page

```

This example will produce the following HTML code.

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html>
<head>
<meta http-equiv="Content-Type"
      content="text/html; charset=iso-8859-1">
<title>An HTML page example</title>
<meta name="author" content="The doc author">
<link href=/style.css rel="stylesheet"
      type="text/css">
</head>
<body>
<p class="title">Hello World
</body>
</html>

```

7.3 Writing CGI scripts

With an HTML page generator and a query string parser, **Aleph** is armed to handle *Common Gateway Interface* or CGI scripts. The rule of the game are quite simple. Everything rely on the request method and the protocol in use. We illustrate this by first looking at the *query string* retrieval, then parsing it, and finally generating the HTML result or an error status.

7.3.1 Getting the query string

If the request method is GET, then the query string is available in the environment variable QUERY_STRING. If the request method is POST, the query string is available in the input stream. The length of the query string is given by the CONTENT_LENGTH environment variable. The following example illustrates the extraction of the query string.

```

# initialize the query result
const query (aleph:sys:get-env "QUERY_STRING")
# get the request method
const rqm (aleph:sys:get-env "REQUEST_METHOD")
# check for a post request
if (== rqm "POST") {
  # create a buffer from the content length

```

```

const len (Integer (aleph:sys:get-env "CONTENT_LENGTH"))
# get the standard input stream and read content
const is (interp:get-input-stream)
const buf (is:read len)
# set the query string
query:= (buf:to-string)
}

```

7.3.2 Parsing the query string

The **CgiQuery** class is designed to parse a CGI query string. Once the string has been parsed, it is possible to perform a query by key. The class takes care of converting all special characters as described in the URL class. The class is constructed with the query string as an argument.

```

const query (aleph:www:CgiQuery "name=hello&value=world")
query:length      → 2
query:get "name"  → hello
query:get "value" → world

```

Armed with the `length` and `get` methods, one can enjoy to write nice CGI script. Note that the class provides numerous methods to query for the existence of a key, get a key or a value by index and many more. All of them are described in the volume 2, *Library reference manual*.

7.3.3 A complete example

We illustrate our discussion with a simple CGI script which prints the value of an environment variable. We assume that the request method is GET. We assume as well that an HTML page uses a simple *input* form to query the name the environment variable to query. The key will be denoted name to refer to the user input.

```

# get the libraries
interp:library "aleph-sys"
interp:library "aleph-www"

# extract the query string
const query (aleph:www:CgiQuery
              (aleph:sys:get-env "QUERY_STRING"))
# get the environment value
const name (query:get "name")
const value (aleph:sys:get-env name)

# print the result embedded in a simple html page
const page (aleph:www:HtmlPage)
page:set-title "Environment variable query result"
page:add-body "<p>Query result for: " name " = " value
page:write-cgi

```

This is it. That's all what is needed. The `write-cgi` method take care of responding to the HTTP server by specifying the status and the result content type (aka `text/html`).

7.4 Cookie

The **Cookie** object is a special object that can be used during a CGI session, to post data to the HTTP client. The idea behind *cookies* is to be able to maintain some state, during the user session of for some time. A cookie is a *name/value* pair and eventually an expiration time. By default, the **Aleph** cookie object are defined for one HTTP client session, but this behavior can be changed.

7.4.1 Managing cookies

A cookie is created with a *name/value* pair and eventually an expiration time. Such expiration time is called the *maximum-age* and is automatically formatted by the object. With two arguments a session cookie is created. With a third argument as an integer, the constructor set the maximum age in second since in reference to the current time.

```
# create a cookie with name/value
const cookie (aleph:www:Cookie "cartid" "12345678")
```

The cookie method support the RFC 2109 which is actually obsoleted by RFC 2965. However, both RFC are not widely supported, so the original cookie definition has been implemented by default. This means that most of the available method have no effect. Most of the time, the methods used are `set-max-age` and `set-path`. A `set-domain` method is also available but its use is not recommended since most the HTTP client have a security mechanism where only cookies originating from the same domain are accepted.

The `set-max-age` method sets the cookie life time in seconds, in reference to the current time. A negative value is always reset to -1 and defined a session cookie. a 0 value tells the HTTP client to remove the cookie. The `set-path` method defines the path for which this cookie apply.

7.4.2 Adding a cookie

Once the cookie is defined, the `set-cookie` method of the **HtmlPage** object can be used to install the cookie. Combined with the `write-cgi` method, the complete page can be send to the HTTP client. For illustration, we provide the code of the *aleph web site* that demonstrates the usage of a cookie.

```
# get the aleph libraries
interp:library "aleph-sys"
interp:library "aleph-www"

# load the html utils
interp:load "html-util.als"

# get the query string
const query (get-query)
const cname (query:get "name")
const value (query:get "value")
const maxtm (query:get "maxtm")

# prepare the html page
const page (aleph:www:HtmlPage)

# check if we add the cookie
try {
  if (!= (maxtm:length) 0) {
    const tmval (Integer maxtm)
    const cookie (aleph:www:Cookie cname value tmval)
```

```

    } {
        const cookie (aleph:www:Cookie cname value)
    }
    # add the cookie to the page
    page:add-cookie cookie
}

# add a simple message
add-page-title page "Aleph Cookie Tester"
# set some news
page:add-body "<p class=text>"
page:add-body "Thank you for using the cookie tester."
page:add-body "You can check the cookie setting by using the"
page:add-body "<a href=/cgi/browser.als>browser demo</a>"
page:add-body "The cookie is stored in the HTTP_COOKIE"
page:add-body "variable. You can also go back to the"
page:add-body " <a href=/cgi/example.als>example</a> page."
page:add-body "<br><br>"
page:add-body "<div class=example><pre>"
page:add-body "Cookie name   : " cname
page:add-body "Cookie value : " value
page:add-body "</pre></div>"
# add the footer
page:add-body "<p>"
add-page-footer page

# write the html page
page:write-cgi

```

CHAPTER 8

Introduction

The **Aleph Object database** or **AOD** system is an environment that integrates data in such way that they can be further processed. The **AOD** system is divided in two parts. One is the programming environment called **ODB**. The other one is an interface that uses the **ODB** programming environment. **ODB** is built on top of the **Aleph Programming Language**. **AOD** is built on top of **ODB**. **AOD** can also be seen as a command environment that permits to administer the database.

8.1 Data integration

The sole purpose of performing *data integration* is to collect various data and store them in such a way that they can be accessed later. Unlike standard *Database environment*, **AOD** does not place restrictions on the data organization. Although, the concept of tables is still present, there is no requirement concerning the number of columns, columns names or similar things. **AOD** operates *on demand*, to integrate data.

8.2 Basic concepts

The **AOD** system integrates data in a hierarchical fashion. The basic data element is called a *cell*. A set of cell is a *record*. A set of records is a *table*. A set of tables and records is *collection*.

8.2.1 Cell and data

A **cell** is a data container. There is only one data element per cell. Eventually a name can be associated with a cell. The cell data can be any kind of literals. Such literals are integer, real, boolean, character or strings.

8.2.2 Record

A **record** is a vector of cells. A record can be created by adding cell or simply by adding data. If the record has a predefined size, the cell or data can be set by indexing (i.e position).

8.2.3 Table

A **table** is a vector of records. A table can be created by adding record. Similarly, if the table has a predefined size, record cell or data can be added by indexing. A table can be also seen as a 2

dimensional array of cells.

8.2.4 Collection

A **collection** is a set of tables and/or records. A collection of table permits to structure data in the form of sheets. Since cell, record and table can have a name, it is possible to create link between various elements, thus creating a collection of structure data.

CHAPTER 9

Integration and Importation

The process of importing data requires first to create a *collectable environment*. When starting from the beginning, the best way is to create a new **Collection** with the `open` command, create one or several tables with the `create` command and finally import data with the `import` command. The `save` command write a binary representation of the collection in a file.

9.1 Creating a collection

Creating a new *collection* is a single operation with the `open` command. The command take a single string argument as the collection name.

```
(aod) aod:open "elements"
```

This command creates a new **Collection** object named `elements`. Once created, the collection becomes the default one for the session. The next step is to create a table that will be used during the importation process.

```
(aod) aod:create "data"
```

A new table named `data` is created and becoming the default one. Once the table is created, data can be added to it; either by importation or by direct integration.

9.1.1 The periodic table of elements

The *Periodic table of elements* is a simple example that illustrates the importation process. The original file is located into the `exp/elem` directory. The `elements.tbl` file is a simple file that associates the *atomic number* with an element name, its chemical symbol and other parameters. An extract of the file is shown below.

```
# -----  
# - elements.tbl -  
# - the periodic table of elements -  
# -----  
# - element name    sym    weight    bp (c)    mp (c)    density    -  
# -----  
1    "Hydrogen"      "H"      1.00797  -252.7    -259.2    0.071  
2    "Helium"        "He"     4.0026   -268.9    -269.7    0.126
```

3	"Lithium"	"Li"	6.939	1330	180.5	0.53
4	"Beryllium"	"Be"	9.0122	2770	1277	1.85
5	"Boron"	"B"	10.811	nil	2030	2.34

There are 112 rows in this table. Not all rows have data. A particular cell, with no data is marked with the special symbol **nil**. Strings are enclosed with double quotes. Integer and reals numbers differentiate themselves automatically. A line with no data is ignored. A comment starts with the '#' character. The importation process is very simple.

```
(aod) aod:open    "elements"
(aod) aod:create  "data"
(aod) aod:import  "elements.tbl"
(aod) aod:save    "elements.odb"
```

After the `import` command, the `save` command write the collection in a file called `elements.orb`. Such file can be later used with the `open` command. The command file `import.als` is an **ODB** script that does the same thing. Note that the script is also an **Aleph** file.

APPENDIX A

Streams

This chapter is a reference of the Aleph input/output streams. The classes described here are part of the **aleph-sio** library. The library must be loaded prior any use of these functions. Once the library is loaded, all functions are located in the **aleph:sio** nameset.

Table 2 Aleph standard input/output streams

Object	Description
Input	input stream base class
Output	output stream base class
Terminal	input/output terminal stream
Selector	i/o multiplexing selector
Directory	directory information
InputFile	input file stream
InputTerm	input terminal stream
InputString	input string stream
InputMapped	input mapped stream
ErrorTerm	error terminal stream
OutputFile	output file stream
OutputTerm	output terminal stream
OutputString	output string stream

For each stream object, a predicate is provided.

Table 3 Aleph stream object predicates

Object	Predicate
input stream	input-p
output stream	output-p
Terminal	terminal-p
Directory	directory-p
InputFile	input-file-p
InputTerm	input-term-p
InputString	input-string-p
InputMapped	input-mapped-p
OutputFile	output-file-p
OutputTerm	output-term-p
OutputString	output-string-p
Selector	selector-p

Input [aleph:sio]

Description

The `Input` class is a base class for the **Aleph** standard input/output library. The class is automatically constructed by a derived class and provide the common methods for all input streams.

Methods Summary

Method	Description
<code>read</code>	returns the next available character
<code>readln</code>	returns the next available line
<code>valid-p</code>	returns true if a character is available
<code>eof-p</code>	returns true if the file is at its end
<code>pushback</code>	pushback a character or a string
<code>get-buffer-length</code>	return the length of the pushback buffer

Input:read

- return: Character
- arguments: none

The **read** method returns the next character available from the input stream. If the stream has been closed or consumed, the `eof` character is returned.

Input:readn

- return: Buffer
- arguments: Integer

The **readn** method returns a buffer object with at most the number of characters specified as an argument. The `buffer length` method should be used to check how many characters have been placed in the buffer.

Input:readln

- return: String
- arguments: none

The **readln** method returns the next line available from the input stream. If the stream has been closed or consumed, the `eof` character is returned.

Input:valid-p

- return: Boolean
- arguments: none | Integer

The **valid-p** method returns `true` if the input stream is in a valid state. By valid state, we mean that the input stream can return a character with a call to the `read` method. With one argument, the method times out after the specified time in milliseconds. If the timeout is null, the method returns immediately. With -1, the method blocks indefinitely if no character is available.

Input:eof-p

- return: Boolean
- arguments: none

The **eof-p** method returns `true` if the input stream has been closed or consumed.

Input:pushback

- return: none
- arguments: Character | String

The **pushback** method pushback a character or a string in the input stream. Subsequent calls to `read` will return the last pushed characters. Pushing a string is equivalent to push each characters of the string.

Input:get-buffer-length

- return: Integer
- arguments: none

The **get-buffer-length** method returns the length of the pushback buffer.

InputFile [aleph:sio]

Description

The `InputFile` class provide the facility for an input file stream. An input file instance is created with a file name. If the file does not exist or cannot be opened, an exception is raised. The `InputFile` class is derived from the `Input` class.

Constructors Summary

Constructor	Description
<code>InputFile file-name</code>	create an input file by name

Derivation summary

Derived from	Description
<code>Input</code>	the input stream class

Methods Summary

Method	Description
<code>lseek</code>	set the file at a certain position
<code>close</code>	close this input file
<code>length</code>	return the length of the input file
<code>get-name</code>	returns the input file name

InputFile:get-name

- return: `String`
- arguments: `none`

The **get-name** method returns the input file name.

InputFile:close

- return: `Boolean`
- arguments: `none`

The **close** method close the input file and returns `true` on success, `false` otherwise. In case of success, multiple calls return `true`.

InputFile:lseek

- return: none
- arguments: Integer

The **lseek** set the input file position to the integer argument. Note that the pushback buffer is reset after this call.

InputFile:length

- return: Integer
- arguments: none

The **length** method returns the length of the input file. The length is expressed in characters.

InputMapped [aleph:sio]

Description

The `InputMapped` class provide the facility for an input file stream with offset and size. The class is similar to the `InputFile` class except that the constructor can also accepts an integer offset and size argument. If the file offset or size are out of range, the class behaves like an input file. If the file does not exist or cannot be opened, an exception is raised. The `InputMapped` class is derived from the `Input` class.

Constructors Summary

Constructor	Description
<code>InputMapped file-name</code>	create a mapped input file
<code>InputMapped file-name offset size</code>	create a mapped input file

Derivation summary

Derived from	Description
<code>Input</code>	the input stream class

Methods Summary

Method	Description
<code>lseek</code>	set the mapped file at a certain position
<code>length</code>	return the length of the mapped file
<code>get-name</code>	returns the mapped file name
<code>get-offset</code>	returns the mapped file offset

InputMapped:lseek

- return: none
- arguments: Integer

The **`lseek`** set the input mapped file position to the integer argument. Note that the pushback buffer is reset after this call.

InputMapped:length

- return: Integer

■ arguments: none

The **length** method returns the length of the input mapped file. The length is expressed in characters.

InputMapped:get-name

■ return: String

■ arguments: none

The **get-name** method returns the input mapped file name.

InputMapped:get-offset

■ return: Integer

■ arguments: none

The **get-name** method returns the input mapped file offset.

InputString [aleph:sio]

Description

The `InputString` class provide the facility for an input string stream. The class is initialized or set with a string and then behaves like a stream. This class is very usefull to handle generic stream method without knowing what kind of stream is behind it.

Constructors Summary

Constructor	Description
<code>InputString</code>	create an empty input string
<code>InputString value</code>	create an input string by value

Derivation summary

Derived from	Description
<code>Input</code>	the input stream class

Methods Summary

Method	Description
<code>set</code>	set the input string value
<code>get</code>	get a character from the stream

InputString:get

- return: `Character`
- arguments: `none`

The `get` method returns the next available character from the input stream but do not remove it.

InputString:set

- return: `none`
- arguments: `String`

The `set` method sets the input string by first resetting the pushback buffer and then initializing the input string with the argument value.

InputTerm [aleph:sio]

Description

The `InputTerm` class provide the facility for an input terminal stream. The input terminal reads character from the standard input stream. No line editing facility is provided with this class This is a low level class, and normally, the **Terminal** class should be used instead.

Constructors Summary

Constructor	Description
<code>InputTerm</code>	create an input terminal

Derivation summary

Derived from	Description
<code>Input</code>	the input stream class

Methods Summary

Method	Description
<code>set-eof-ignore</code>	set the ctrl-d ignore flag
<code>set-eof-character</code>	set the ctrl-d character

InputTerm:set-eof-ignore

- return: none
- arguments: Boolean

The **set-eof-ignore** method set the input terminal *ctrl-d* ignore flag. When the flag is on, any character that match a ctrl-d is changed to the remapped character and returned by a read. This method is usefull to prevent a reader to exit when the *ctrl-d* character is generated.

InputTerm:set-eof-character

- return: none
- arguments: Character

The **set-eof-character** method set the input terminal *ctrl-d* remapping character. By default the character is set to the *end-of-line* character. This method should be used in conjunction with the `set-eof-ignore` method.

OutputFile [aleph:sio]

Description

The `Output` class is a base class for the **Aleph** standard input/output library. The class is automatically constructed by a derived class and provide the common methods for all output streams.

Methods Summary

Method	Description
<code>write</code>	write literals
<code>writeln</code>	write literals followed by a newline
<code>errorln</code>	write literals followed by a newline
<code>newline</code>	write a new line character

Output:write

- return: none
- arguments: [Literal...]

The **write** method write one or more literal arguments on the output stream. This method returns nil;

Output:writeln

- return: none
- arguments: [Literal...]

The **writeln** method write one or more literal argument to the output stream and finish with a newline. This method return nil.

Output:errorln

- return: none
- arguments: [Literal...]

The **errorln** method write one or more literal argument to the associated output error stream and finish with a newline. Most of the time, the output stream and error stream are the same except for an output terminal.

Output:newline

- return: none
- arguments: none

The **newline** method writes a new line character to the output stream. The method returns nil.

OutputFile [aleph:sio]

Description

The `OutputFile` class provide the facility for an output file stream. An output file instance is created with a file name. If the file does not exist, it is created. If the file cannot be created, an exception is raised. Once the file is created, it is possible to write literals. The class is derived from the `Output` class. By default an output file is created if it does not exist. If the file already exist, the file is truncated to 0. Another constructor for the output file gives more control about this behavior. It takes two boolean flags that defines the truncate and append mode. The `t-flag` is the truncate flag. The `a-flag` is the append flag.

Constructors Summary

Constructor	Description
<code>OutputFile file-name</code>	create an output file by name
<code>OutputFile file-name t-flag a-flag</code>	create an output file by name and flag

Derivation summary

Derived from	Description
<code>Output</code>	the output stream class

Methods Summary

Method	Description
<code>get-name</code>	returns the output file name
<code>close</code>	close this output file

OutputFile:close

- return: Boolean
- arguments: none

The `close` method closes the output file and returns `true` on success, `false` otherwise. In case of success, multiple calls return `true`.

OutputFile:get-name

- return: String

■ arguments: none

The **get-name** method returns the output file name.

OutputString [aleph:sio]

Description

The `OutputString` class provide the facility for an output string stream. The class is initially empty and acts as a buffer which accumulate the write method characters. The `to-string` method can be used to retrieve the buffer content.

Constructors Summary

Constructor	Description
<code>OutputString</code>	create an empty output string
<code>OutputString value</code>	create an output string with a value

Derivation summary

Derived from	Description
<code>Output</code>	the output stream class

Methods Summary

Method	Description
<code>flush</code>	flush the output string
<code>to-string</code>	return the string buffer

OutputString:flush

- return: none
- arguments: none

The **flush** method flushes the output string stream by resetting the string buffer.

OutputString:to-string

- return: String
- arguments: none

The **to-string** method returns a string representation of the output string buffer.

OutputTerm [aleph:sio]

Description

The `OutputTerm` class provide the facility for an output terminal. The output terminal is defined as the standard output stream. If the standard error stream needs to be used, the **ErrorTerm** class is more appropriate. The class is derived from the `Output` class.

Constructors Summary

Constructor	Description
ErrorTerm	create an error terminal
OutputTerm	create an output terminal

Derivation summary

Derived from	Description
Output	the output stream class

Terminal [aleph:sio]

Description

The `Terminal` class provide the facility for an input/output terminal with line editing capability. The class combines the **InputTerm** and **OutputTerm** methods.

Constructors Summary

Constructor	Description
<code>Terminal</code>	create a new terminal

Derivation summary

Derived from	Description
<code>InputTerm</code>	the input terminal class
<code>OutputTerm</code>	the output terminal class

Methods Summary

Method	Description
<code>set-primary</code>	set the primary prompt
<code>set-secondary</code>	set the secondary prompt
<code>get-primary</code>	get the primary prompt
<code>get-secondary</code>	get the secondary prompt

Terminal:set-primary

- return: `none`
- arguments: `String`

The **set-primary** method sets the terminal primary prompt which is used when the `readline` method is called.

Terminal:set-secondary

- return: `none`
- arguments: `String`

The **set-secondary** method sets the terminal secondary prompt which is used when the `readline` method is called.

Terminal:get-primary

- return: String
- arguments: none

The **get-primary** method returns the terminal primary prompt.

Terminal:get-secondary

- return: String
- arguments: none

The **get-secondary** method returns the terminal secondary prompt.

Directory [aleph:sio]

Description

The `Directory` class provides some facilities to access a directory. By default, a directory object is constructed to represent the current directory. With one argument, the object is constructed from the directory name. Once the object is constructed, it is possible to retrieve its content.

Constructors Summary

Constructor	Description
<code>Directory</code>	open the current directory
<code>Directory <i>directory-name</i></code>	open a directory by name

Methods Summary

Method	Description
<code>mkdir</code>	create a directory
<code>rmdir</code>	remove a directory
<code>rmfile</code>	remove a file
<code>get-name</code>	return the directory name
<code>get-list</code>	return a list of the directory contents
<code>get-files</code>	return a list of files in this directory
<code>get-subdirs</code>	return a list of sub directories

Directory:mkdir

- return: `Directory`
- arguments: `String`

The `mkdir` method creates a new directory in the current one. The full path is constructed by taking the directory name and adding the argument. Once the directory is created, the method returns a directory object of the newly constructed directory. An exception is thrown if the directory cannot be created.

Directory:rmdir

- return: `none`
- arguments: `String`

The `rmdir` method removes an empty directory. The full path is constructed by taking the directory name and adding the argument. An exception is thrown if the directory cannot be removed.

Directory:rmfile

- return: none
- arguments: String

The **rmfile** method removes a file in the current directory. The full path is constructed by taking the directory name and adding the argument. An exception is thrown if the file cannot be removed.

Directory:get-name

- return: String
- arguments: none

The **get-name** method returns the directory name. If the default directory was created, the method returns the full directory path.

Directory:get-list

- return: List
- arguments: none

The **get-list** method returns the directory contents. The method returns a list of strings. The list contains all valid names at the time of the call, including the current directory and the parent directory.

Directory:get-files

- return: List
- arguments: none

The **get-files** method returns the directory contents. The method returns a list of strings of files. The list contains all valid names at the time of the call.

Directory:get-subdirs

- return: List
- arguments: none

The **get-subdirs** method returns the sub directories. The method returns a list of strings of subdirectories. The list contains all valid names at the time of the call, including the current directory and the parent directory.

Selector [aleph:sio]

Description

The `Selector` class provides some facilities to perform I/O multiplexing. The constructor takes 0 or several stream arguments. The class manages automatically to differentiate between **Input** and **Output** streams. Once the class is constructed, it is possible to get the first stream ready for reading or writing or all of them. It is also possible to add more streams after construction with the `add` method. When used with several input streams in a multi-threaded context, the selector behavior can become quite complicated. Either `wait` and `wait-all` methods check first the input streams push-back buffer. If one or several buffer is (are) not empty, the method returns with these streams. During this operation, the input streams are locked, so no other thread can push-back a character. The selector then checks for status change and unlock the streams. Note that the output streams are not locked. Note also that a thread which relies on the input stream push-back method to release a selector will result in a dead lock.

Constructors Summary

Constructor	Description
<code>Selector</code>	create an empty selector
<code>Selector [Input/Output]</code>	create a selector with streams

Methods Summary

Method	Description
<code>add</code>	add a new stream to the selector
<code>wait</code>	wait for one stream to change status
<code>wait-all</code>	wait for some stream to change status
<code>input-get</code>	return an input stream by index
<code>output-get</code>	return an output stream by index
<code>input-length</code>	return the number of input streams
<code>output-length</code>	return the number of output streams

Selector:add

- return: `nil`
- arguments: `Input | Output`

The `add` method adds an input or output stream to the selector.

Selector:wait

- return: `Object`

■ arguments: none | Integer

The **wait** method waits for a status change in the selector and returns the first stream that has change status. With one argument, the selector time-out after the specified time in milliseconds. Note that at the time of the return, several streams may have changed status.

Selector:wait-all

■ return: Vector

■ arguments: none | Integer

The **wait** method waits for a status change in the selector and returns all streams that has change status in a vector object. With one argument, the selector time-out after the specified time in milliseconds. If the selector has timed-out, the vector is empty.

Selector:input-get

■ return: Input

■ arguments: Integer

The **input-get** method returns the input streams in the selector by index. If the index is out of bound, an exception is raised.

Selector:output-get

■ return: Output

■ arguments: Integer

The **output-get** method returns the output streams in the selector by index. If the index is out of bound, an exception is raised.

Selector:input-length

■ return: Integer

■ arguments: none

The **input-length** method returns the number of input streams in the selector.

Selector:output-length

■ return: Integer

■ arguments: none

The **output-length** method returns the number of output streams in the selector.

APPENDIX B

File System Functions

This chapter is a reference of the Aleph file system functions. The functions described here are part of the **aleph-sio** library. The library must be loaded prior any use of these functions. Once the library is loaded, all functions are located in the **aleph:sio** nameset.

Table 4 Aleph file system functions

Object	Description
dir-p	check for a directory
file-p	checks for a regular file
rmdir	remove one or several directories
rmfile	remove one or several files
absolute-path	create an absolute path name
relative-path	create a relative path name

aleph:sio:dir-p

- return: Boolean
- arguments: String

The **dir-p** function returns true if the argument name is a directory name, false otherwise.

aleph:sio:file-p

- return: Boolean
- arguments: String

The **file-p** function returns true if the argument name is a regular file name, false otherwise.

aleph:sio:absolute-path

- return: String
- arguments: [String ...]

The **absolute-path** function returns an absolute path name from an argument list. Without argument, the command returns the root directory name. With one or several argument, the absolute path is computed from the root directory.

aleph:sio:relative-path

- return: String
- arguments: [String ...]

The **relative-path** function returns a relative path name from an argument list. With one argument, the function returns it. With two or more arguments, the relative path is computed by joining each argument with the previous one.

aleph:sio:rmfile

- return: none
- arguments: [String ...]

The **rmfile** function removes one or several files specified as the arguments. If one file fails to be removed, an exception is raised.

aleph:sio:rmdir

- return: none
- arguments: [String ...]

The **rmdir** function removes one or several directories specified as the arguments. If one directory fails to be removed, an exception is raised.

APPENDIX C

System Classes

This chapter is a reference of the Aleph system classes. The classes described here are part of the **aleph-sys** library. The library must be loaded prior any use of these classes. Once the library is loaded, all classes are located in the **aleph:sys** nameset.

Table 5 Aleph system classes

Object	Description
Time	time and date class

For each system class, a predicate is provided.

Table 6 Aleph system class predicates

Object	Predicate
Time	time-p

Time [aleph:sys]

Description

The `Time` class is the system access to the date and time. When an instance of that class is created, the creation time is recorded in the instance. The recorded time is the *epoch* time corresponding to the UTC time of January 1, 1970. The resolution is in seconds. Various methods are provided to extract the date and time. The time can either be the local time or the UTC time. With one argument, the object is initialized to the date specified as an integer argument in reference to the *epoch*.

Constructors Summary

Constructor	Description
<code>Time</code>	create a new time class
<code>Time tval</code>	create a new time class by value

Methods Summary

Method	Description
<code>add</code>	add a time in second to the current time
<code>get-time</code>	returns the time in seconds since the epoch
<code>get-year</code>	returns the local year
<code>get-hours</code>	returns the local number of hours
<code>get-seconds</code>	returns the local number of seconds
<code>get-minutes</code>	returns the local number of minutes
<code>get-day-of-week</code>	returns the local day in the week
<code>get-day-of-year</code>	returns the local day in the year
<code>get-day-of-month</code>	returns the local day in the month
<code>get-month-of-year</code>	returns the local month in the year
<code>get-utc-year</code>	returns the utc year
<code>get-utc-hours</code>	returns the utc number of hours
<code>get-utc-seconds</code>	returns the utc number of seconds
<code>get-utc-minutes</code>	returns the utc number of minutes
<code>get-utc-day-of-week</code>	returns the utc day in the week
<code>get-utc-day-of-year</code>	returns the utc day in the year
<code>get-utc-day-of-month</code>	returns the utc day in the month
<code>get-utc-month-of-year</code>	returns the utc month in the year
<code>format-date</code>	format local date
<code>format-time</code>	format local time
<code>utc-format-date</code>	format utc date
<code>utc-format-time</code>	format utc time
<code>utc-format-rfc</code>	format utc time as RFC 822
<code>utc-format-cookie</code>	format utc time for cookie expire

Time:add

- return: none
- arguments: Integer

The **add** method adds the time argument in second to the current time value. The new date is recomputed after it. This method is useful to compute a time in the future, in reference to the current time.

Time:get-time

- return: Integer
- arguments: none

The **get-time** method returns the number of seconds elapsed since the epoch. The epoch is January 1, 1970.

Time:get-seconds

- return: Integer
- arguments: none

The **get-seconds** method returns the number of seconds after the minute for the local time, corrected for daylight saving time. The returned value is the range 0 to 61, eventually accounting for the leap second.

Time:get-minutes

- return: Integer
- arguments: none

The **get-minutes** method returns the number of minutes after the hour for the local time, corrected for daylight saving time. The returned value is the range 0 to 59.

Time:get-hours

- return: Integer
- arguments: none

The **get-hours** method returns the number of hours since midnight for the local time, corrected for daylight saving time. The returned value is the range 0 to 23.

Time:get-month

- return: Integer
- arguments: none

The **get-month** method returns the month in the year for the local time, corrected for daylight saving time. The returned value is the range 0 to 11.

Time:get-year

- return: Integer

■ arguments: none

The **get-year** method returns the year for the local time, corrected for daylight saving time. The returned value is an absolute year value (year 2000 is 2000).

Time:get-day-of-month

■ return: Integer

■ arguments: none

The **get-day-of-month** method returns the day in the month for the local time, corrected for daylight saving time. The returned value is the range 1 to 31.

Time:get-day-of-week

■ return: Integer

■ arguments: none

The **get-day-of-week** method returns the day in the week for the local time, corrected for daylight saving time. The returned value is the range 0 to 6 in reference to Sunday.

Time:get-day-of-year

■ return: Integer

■ arguments: none

The **get-day-of-year** method returns the day in the year for the local time, corrected for daylight saving time. The returned value is the range 0 to 365 in reference to January 1.

Time:get-utc-seconds

■ return: Integer

■ arguments: none

The **get-utc-seconds** method returns the number of seconds after the minute for the UTC time. The returned value is the range 0 to 61, eventually accounting for the leap second.

Time:get-utc-minutes

■ return: Integer

■ arguments: none

The **get-utc-minutes** method returns the number of minutes after the hour for the UTC time. The returned value is the range 0 to 59.

Time:get-utc-hours

■ return: Integer

■ arguments: none

The **get-utc-hours** method returns the number of hours since midnight for the UTC time. The returned value is the range 0 to 23.

Time:get-utc-month

- return: Integer
- arguments: none

The **get-utc-month** method returns the month in the year for the UTC time. The returned value is the range 0 to 11.

Time:get-utc-year

- return: Integer
- arguments: none

The **get-utc-year** method returns the year for the UTC time, The returned value is an absolute year value (year 2000 is 2000).

Time:get-utc-day-of-month

- return: Integer
- arguments: none

The **get-utc-day-of-month** method returns the day in the month for the UTC time. The returned value is the range 1 to 31.

Time:get-utc-day-of-week

- return: Integer
- arguments: none

The **get-utc-day-of-week** method returns the day in the week for the UTC time. The returned value is the range 0 to 6 in reference to Sunday.

Time:get-utc-day-of-year

- return: Integer
- arguments: none

The **get-utc-day-of-year** method returns the day in the year for the UTC time. The returned value is the range 0 to 365 in reference to January 1.

Time:format-date

- return: String
- arguments: none

The **format-date** method returns a formatted string of the local date.

Time:format-time

- return: String
- arguments: none

The **format-time** method returns a formatted string of the local time.

Time:utc-format-date

- return: String
- arguments: none

The **utc-format-date** method returns a formatted string of the utc date.

Time:utc-format-time

- return: String
- arguments: none

The **utc-format-time** method returns a formatted string of the utc time.

Time:utc-format-rfc

- return: String
- arguments: none

The **utc-format-rfc** method returns a formatted string of the utc date and time as specified by RFC 822.

Time:utc-format-cookie

- return: String
- arguments: none

The **utc-format-cookie** method returns a formatted string of the utc date and time suitable to be used as a cookie expiration time. Of course, the cookie time is not the same as the RFC 822.

APPENDIX D

System Calls

This chapter is a reference of the Aleph system calls. The functions described here are part of the **aleph-sys** library. The library must be loaded prior any use of these functions. Once the library is loaded, all functions are located in the **aleph:sys** nameset.

Table 7 Aleph system call functions

Function	Description
exit	exit unconditionally with an exit code
sleep	pause for a certain time
random	return a random integer number
get-pid	get the process identifier
get-env	get an environment variable
get-host-name	return the host name
get-user-name	return the user name

aleph:sys:exit

- return: none
- arguments: Integer

The **exit** function exit unconditionally with the exit code as the argument.

aleph:sys:sleep

- return: none
- arguments: Integer

The **sleep** function pause the specific thread for a certain time. The time argument is expressed in milliseconds. This function returns nil.

aleph:sys:random

- return: Integer
- arguments: none

The **random** function returns a random integer number. The function is protected by a mutex. A calling thread will block until the other one has completed the call.

aleph:sys:get-pid

- return: Integer
- arguments: none

The **get-pid** function returns the process identifier (pid). The returned value is a positive integer.

aleph:sys:get-env

- return: String
- arguments: String

The **get-env** function returns the environment variable associated with the string argument. If the environment does not exist an exception is raised.

aleph:sys:get-host-name

- return: String
- arguments: none

The **get-host-name** function returns the host name. The host name can be either a simple name or a canonical name with its domain, depending on the system configuration.

aleph:sys:get-user-name

- return: String
- arguments: none

The **get-user-name** function returns the current user name.

APPENDIX E

Formatting

This appendix is a reference of the Aleph text processing library. All classes described here are part of the **aleph-txt** library. The library must be loaded prior any use of these classes. Once the library is loaded, all classes are located in the **aleph:txt** nameset.

Table 8 Aleph text processing classes

Object	Description
PrintTable	table formatting object
Digest	message digest computation

For each class, a predicate is provided.

Table 9 Aleph text processing predicates

Object	Predicate
PrintTable	print-table-p
Digest	digest-p

PrintTable [aleph:txt]

Description

The `PrintTable` class is a formatting class for tables. The table is constructed with the number of columns (default to 1) and eventually the number of rows. Once the table is created, element are added to the table with the `add` method. Specific table element can be set with the `set` method. The class provide a `format` method those default is to print the table on the interpreter standard output. With an output stream argument or a buffer, the table is formatted to these objects. The table formatting includes an optional column width, a filling character and a filling direction flag. By default, the column width is 0. This means that the column width is computed as the maximum length of all column elements. If the column width is set with the `set-column-size` method, the string element might be truncated to the left or right (depending on the filling flag) to fit the column width.

Derivation summary

Derived from	Description
Object	the base object class

Constructors Summary

Constructor	Description
<code>PrintTable</code>	create a one column table
<code>PrintTable cols</code>	create a multi-column table
<code>PrintTable cols rows</code>	create a multi-column table with rows

Methods Summary

PrintTable:add

- return: none | Integer
- arguments: none | [Literal...]

The `add` method serves several purposes. Without argument, a new row is added and the row index is returned. The row index can be later used with the `set` method to set a particular table element. With one or several literal arguments (those length must match the number of columns), a new row is created and those arguments added to the table. In that later case, the method returns the nil object.

PrintTable:get

Method	Description
add	add a new column and eventually element
get	get a table element by row and column
set	set a table element by row and column
format	format the table to a stream or buffer
get-rows	return the number of rows
get-columns	return the number of columns
set-column-size	set a column desired size
get-column-size	get a column desired size
set-column-fill	set a column fill character
get-column-fill	get a column fill character
set-column-direction	set a column fill direction flag
get-column-direction	get a column fill direction flag

■ return: String

■ arguments: Integer Integer

The **get** method returns a particular table element by row and column. The first argument is the table row index and the second is the table column index.

PrintTable:set

■ return: none

■ arguments: Integer Integer Literal

The **set** method sets a particular table element by row and column. The first argument is the table row index and the second is the table column index. The last argument is a literal object that is converted to a string prior its insertion.

PrintTable:format

■ return: none

■ arguments: none | Output | Buffer

The **format** method writes the formatted table to an output stream or a buffer. Without argument, the default interpreter output stream is used.

PrintTable:get-rows

■ return: Integer

■ arguments: none

The **get-rows** method returns the number of rows in the table.

PrintTable:get-columns

■ return: Integer

■ arguments: none

The **get-columns** method returns the number of columns in the table.

PrintTable:set-column-size

- return: none
- arguments: Integer Integer

The **set-column-size** method sets the desired width for a particular column. The first argument is the column index and the second argument is the column width. If 0 is given, the column width is computed as the maximum of the column elements.

PrintTable:get-column-size

- return: Integer
- arguments: Integer

The **get-column-size** method returns the desired width for a particular column.

PrintTable:set-column-fill

- return: none
- arguments: Integer Character

The **set-column-fill** method sets the filling character for a particular column. The first argument is the column index and the second argument is a character to use when filling a particular column element. The default filling character is the blank character.

PrintTable:get-column-fill

- return: Character
- arguments: Integer

The **get-column-fill** method returns the filling character for a particular column.

PrintTable:set-column-direction

- return: none
- arguments: Integer Boolean

The **set-column-direction** method sets the direction flag for a particular column. The first argument is the column index and the second argument is a boolean. A false value indicates a filling by the left while a true value indicates a filling by the right. The column filling character is used for this operation.

PrintTable:get-column-direction

- return: Boolean
- arguments: Integer

The **get-column-direction** method returns the direction flag for a particular column.

Digest [aleph:txt]

Description

The `Digest` class is a message digest computation class. By default, the MD5 algorithm as defined by RFC 1321 is bound to the class. The message digest class computes a *message digest* from an input string or a buffer. The message digest is returned as a string.

Derivation summary

Derived from	Description
Object	the base object class

Constructors Summary

Constructor	Description
Digest	create a default message digest

Methods Summary

Method	Description
compute	compute the message digest

Digest:compute

- return: String
- arguments: String|Buffer

The **compute** method computes a message digest from the input string or the input buffer. The method returns the message digest as a string.

APPENDIX F

Sorting and searching

This chapter is a reference of the **Aleph** sorting and searching functions. The functions described here are part of the **aleph-txt** library. The library must be loaded prior any use of these functions. Once the library is loaded, all functions are located in the **aleph:txt** nameset.

Table 10 Sorting and searching functions

Object	Description
sort	sort a vector

aleph:txt:sort

- return: none
- arguments: Vector

The **sort** function sorts in ascending the vector. The vector is sorted in place.

APPENDIX G

Networking Classes

This appendix is a reference of the Aleph networking services. All classes described here are part of the **aleph-net** library. The library must be loaded prior any use of these classes. Once the library is loaded, all classes are located in the **aleph:net** nameset.

Table 11 Aleph network system classes

Object	Description
Mail	message delivery class
Socket	base socket class
Address	ip address manipulation class
Datagram	udp datagram class
TcpSocket	base tcp socket
TcpClient	tcp client socket class
TcpServer	tcp server socket class
UdpSocket	base udp socket
UdpClient	udp client socket class
UdpServer	udp server socket class
Multicast	multicast socket class

For each class, a predicate is provided.

Table 12 Aleph network class predicates

Object	Predicate
Mail	mail-p
Socket	socket-p
Address	address-p
Datagram	datagram-p
TcpSocket	tcp-socket-p
TcpServer	tcp-server-p
TcpClient	tcp-client-p
UdpSocket	udp-socket-p
UdpServer	udp-server-p
UdpClient	udp-client-p
Multicast	multicast-p

Address [aleph:net]

Description

The `Address` class is the Internet address manipulation class. The class can be used to perform the conversion between a host name and an IP address. The opposite is also possible. Finally, the class supports both IPv4 and IPv6 address formats.

Constructors Summary

Constructor	Description
Address name	create a address class

Methods Summary

Method	Description
get-name	returns the original address name
get-ip-address	returns the ip address as a string
get-ip-vector	returns the ip address as a vector
get-canonical-name	returns the address canonical name

Address:get-name

- return: `String`
- arguments: `none`

The **get-name** method returns the original name used during the object construction.

Address:get-ip-address

- return: `String`
- arguments: `none`

The **get-ip-address** method returns a string representation of the IP address. The string representation follows the IPv4 or IPv6 preferred format, depending on the internal representation.

Address:get-ip-vector

- return: `Vector`
- arguments: `none`

The **get-ip-vector** method returns a vector representation of the IP address. The vector result follows the IPv4 or IPv6 preferred format, depending on the internal representation.

Address: get-canonical-name

■ return: String

■ arguments: none

The **get-canonical-name** method returns a fully qualified name of the address. The resulting name is obtained by performing a reverse lookup. Note that the name can be different from the original name.

Socket [aleph:net]

Description

The `Socket` class is a base class for the **Aleph** network services. The class is automatically constructed by a derived class and provide some common methods for all socket objects.

Derivation summary

Derived from	Description
Input	the input stream class
output	the output stream class

Constants Summary

Constant	Description
REUSE-ADDRESS	enable address reuse
BROADCAST	enable broadcast packet
DONT-ROUTE	bypass routing table lookup
KEEP-ALIVE	test for connection alive
LINGER	linger on close
RCV-SIZE	receive buffer size
SND-SIZE	send buffer size
HOP-LIMIT	the the maximum hop limit
MULTICAST-LOOPBACK	enable the multicast loopback
MULTICAST-HOP-LIMIT	multicast hop limit option
MAX-SEGMENT-SIZE	max TCP segment size
NO-DELAY	disable the naggle algorithm

Methods Summary

Socket:REUSE-ADDRESS

The **REUSE-ADDRESS** constant is used by the `set-option` method to enable socket address reuse. This option changes the rules that validates the address used by `bind`. It is not recommended to use that option as it decreases TCP reliability.

Socket:BROADCAST

The **BROADCAST** constant is used by the `set-option` method to enable broadcast of packets. This options only works with IPV4 address. The argument is a boolean flag only.

Socket:DONT-ROUTE

Method	Description
bind	bind this socket
connect	connect this socket
ipv6-p	check for ipv6 socket
read	returns the next available character
readln	returns the next available line
write	write a character or a string
writeln	write a string followed by a newline
newline	write a new line character
close	close this socket
valid-p	returns true if a character is available
eof-p	returns true if the socket has been closed
pushback	pushback a character or a string
shutdown	shutdown a connection
get-buffer-length	return the read buffer length
get-socket-address	return the socket address
get-socket-port	return the socket port
get-peer-address	return the peer address
get-peer-port	return the peer port
set-option	set a socket option

The **DONT-ROUTE** constant is used by the `set-option` method to control if a packet is to be sent via the routing table. This option is rarely used with **Aleph**. The argument is a boolean flag only.

Socket:KEEP-ALIVE

The **KEEP-ALIVE** constant is used by the `set-option` method to check periodically if the connection is still alive. This option is rarely used with **Aleph**. The argument is a boolean flag only.

Socket:LINGER

The **LINGER** constant is used by the `set-option` method to turn on or off the lingering on close. If the first argument is `true`, the second argument is the linger time.

Socket:RCV-SIZE

The **RCV-SIZE** constant is used by the `set-option` method to set the receive buffer size.

Socket:SND-SIZE

The **SND-SIZE** constant is used by the `set-option` method to set the send buffer size.

Socket:HOP-LIMIT

The **HOP-LIMIT** constant is used by the `set-option` method to set packet hop limit.

Socket:MULTICAST-LOOPBACK

The **MULTICAST-LOOPBACK** constant is used by the `set-option` method to control whether or not multicast packets are copied to the loopback. The argument is a boolean flag only.

Socket:MULTICAST-HOP-LIMIT

The **MULTICAST-HOP-LIMIT** constant is used by the `set-option` method to set the hop limit for multicast packets.

Socket:MAX-SEGMENT-SIZE

The **MAX-SEGMENT-SIZE** constant is used by the `set-option` method to set the TCP maximum segment size.

Socket:NO-DELAY

The **NO-DELAY** constant is used by the `set-option` method to enable or disable the Naggle algorithm.

Socket:bind

- return: none
- arguments: Integer

The **bind** method binds this socket to the port specified as the argument.

Socket:bind

- return: none
- arguments: Integer Address

The **bind** method binds this socket to the port specified as the first argument and the address specified as the second argument.

Socket:connect

- return: none
- arguments: Integer Address

The **connect** method connects this socket to the port specified as the first argument and the address specified as the second argument. A connected socket is useful with udp client that talks only with one fixed server.

Socket:shutdown

- return: Boolean
- arguments: Boolean

The **shutdown** method shutdowns one side of the connection. If the mode argument is false, further receive is disallowed. If the mode argument is true, further send is disallowed. The method returns true on success, false otherwise.

Socket:ipv6-p

- return: Boolean
- arguments: none

The **ipv6-p** predicate returns true if the socket address is an IPV6 address, false otherwise.

Socket:read

- return: Character
- arguments: none

The **read** method returns the next character available from the socket. If the socket has been closed, the `eof` character is returned.

Socket:read

- return: Buffer
- arguments: Integer

The **read** method with an integer argument returns a buffer of characters by reading the socket. The number of read characters might be less than requested. Use the `length` method to check for the returned buffer size.

Socket:readln

- return: String
- arguments: none

The **readln** method returns the next line available from the socket. If the socket has been closed, the `eof` character is returned.

Socket:write

- return: none
- arguments: [Literal...]

The **write** method write one or more literal arguments on the socket. This method returns `nil`;

Socket:writeln

- return: none
- arguments: Literal

The **writeln** method write one or more literal argument to the socket and finish with a newline. This method return `nil`.

Socket:newline

- return: none
- arguments: none

The **newline** method writes a new line character to the socket. The method returns `nil`.

Socket:close

- return: Boolean
- arguments: none

The **close** method close the socket and returns `true` on success, `false` otherwise. In case of success, multiple calls return `true`.

Socket:valid-p

- return: Boolean

■ arguments: [Integer]

The **valid-p** method returns `true` if the socket is in a valid state. By valid state, we mean that the socket can read a character. With one argument, the method timeout after the specified time.

Socket:eof-p

■ return: Boolean

■ arguments: none

The **eof-p** method returns `true` no more characters can be read from this socket or the socket has been closed.

Socket:pushback

■ return: none

■ arguments: Character | String

The **pushback** method pushback a character or a string in the input stream. Subsequent calls to read will return the last pushed characters. Pushing a string is equivalent to push each characters of the string.

Socket:get-socket-address

■ return: Address

■ arguments: none

The **get-socket-address** method returns an address object of the socket. The returned object can be later used to query the canonical name and the ip address.

Socket:get-socket-port

■ return: Integer

■ arguments: none

The **get-socket-address** method returns the port number of the socket.

Socket:get-peer-address

■ return: Address

■ arguments: none

The **get-peer-address** method returns an address object of the socket's peer. The returned object can be later used to query the canonical name and the ip address.

Socket:get-peer-port

■ return: Integer

■ arguments: none

The **get-socket-address** method returns the port number of the socket's peer.

Socket:set-option

- return: Boolean
- arguments: constant [Boolean|Integer] [Integer]

The **set-option** method set a socket option. The first argument is the option to set. The second argument is a boolean value which turn on or off the option. The optional third argument is an integer needed for some options.

TcpSocket [aleph:net]

Description

The `TcpSocket` class is a base class for all tcp socket objects. The class is derived from the `Socket` class and provides some specific tcp methods. If a **TcpSocket** is created, the user is responsible to connect it to the proper address and port.

Constructors Summary

Constructor	Description
<code>TcpSocket</code>	create a new tcp socket

Derivation summary

Derived from	Description
<code>Socket</code>	the socket class

Methods Summary

Method	Description
<code>accept</code>	accept a connection
<code>listen</code>	listen for connection

TcpSocket:accept

- return: `TcpSocket`
- arguments: `none`

The **accept** method waits for incoming connection and returned a **TcpSocket** object initialized with the connected peer. The result socket can be used to perform i/o operations. This method is used by tcp server.

TcpSocket:listen

- return: `Boolean`
- arguments: `none | Integer`

The **listen** method initialize a socket to accept incoming connection. Without argument, the default number of incoming connection is 5. The integer argument can be used to specify the number of incoming connection that socket is willing to queue. This method is used by tcp server.

TcpClient [aleph:net]

Description

The `TcpClient` class creates a tcp client by host and port. The host argument can be either a name or an address object. The port argument is the server port to contact. The **`TcpClient`** class is derived from the **`TcpSocket`** class. This class has no specific methods

Constructors Summary

Constructor	Description
<code>TcpClient</code> <i>host port</i>	create a tcp client by host and port

Derivation summary

Derived from	Description
<code>TcpSocket</code>	the TCP socket class

TcpServer [aleph:net]

Description

The `TcpServer` class creates a tcp server by port. An optional host argument can be either a name or an address object. The port argument is the server port to bind. The **TcpServer** class is derived from the **TcpSocket** class. This class has no specific methods. With one argument, the server bind the port argument on the local host. The backlog can be specified as the last argument. The host name can also be specified as the first argument, the port as second argument and eventually the backlog. Note that the host can be either a string or an address object.

Constructors Summary

Constructor	Description
<code>TcpServer</code>	create a tcp server
<code>TcpServer port</code>	create a tcp server by port
<code>TcpServer port backlog</code>	create a tcp server by port
<code>TcpServer host port</code>	create a tcp server by host
<code>TcpServer host port backlog</code>	create a tcp server by host

Derivation summary

Derived from	Description
<code>TcpSocket</code>	the TCP socket class

Datagram [aleph:net]

Description

The `Datagram` class is a socket class used by `udp` socket. A datagram is constructed by the **UdpSocket** `accept` method. The purpose of a datagram is to store the peer information so one can reply to the sender. The datagram also stores in a buffer the data sent by the peer. This class does not have any constructor nor any specific method.

Derivation summary

Derived from	Description
Socket	the socket class

UdpSocket [aleph:net]

Description

The `UdpSocket` class is a base class for all udp socket objects. The class is derived from the `Socket` class and provides some specific udp methods.

Constructors Summary

Constructor	Description
<code>UdpSocket</code>	create a new udp socket

Derivation summary

Derived from	Description
<code>Socket</code>	the socket class

Methods Summary

Method	Description
<code>accept</code>	accept a datagram

UdpSocket:accept

- return: `Datagram`
- arguments: `none`

The **`accept`** method waits for an incoming datagram and returns a **`Datagram`** object. The datagram is initialized with the peer address and port as well as the incoming data.

UdpClient [aleph:net]

Description

The `UdpClient` class creates a udp client by host and port. The host argument can be either a name or an address object. The port argument is the server port to contact. The **UdpClient** class is derived from the **UdpSocket** class. This class has no specific methods

Constructors Summary

Constructor	Description
<code>UdpClient host port</code>	create a udp client by host and port

Derivation summary

Derived from	Description
<code>UdpSocket</code>	the UDP socket class

UdpServer [aleph:net]

Description

The `UdpServer` class creates a udp server by port. An optional host argument can be either a name or an address object. The port argument is the server port to bind. The **UdpServer** class is derived from the **UdpSocket** class. This class has no specific methods. With one argument, the server bind the port argument on the local host. The host name can also be specified as the first argument, the port as second argument. Note that the host can be either a string or an address object.

Constructors Summary

Constructor	Description
<code>UdpServer</code>	create a udp server
<code>UdpServer port</code>	create a udp server by port
<code>UdpServer host port</code>	create a udp server by host

Derivation summary

Derived from	Description
<code>UdpSocket</code>	the UDP socket class

Multicast [aleph:net]

Description

The `Multicast` class creates a udp multicast socket by port. An optional host argument can be either a name or an address object. The port argument is the server port to bind. The **Multicast** class is derived from the **UdpSocket** class. This class has no specific methods. With one argument, the server bind the port argument on the local host. The host name can also be specified as the first argument, the port as second argument. Note that the host can be either a string or an address object. This class is similar to the **UdpServer** class, except that the socket join the multicast group at construction and leave it at destruction.

Constructors Summary

Constructor	Description
<code>Multicast host</code>	create a multicast socket by host
<code>Multicast addr</code>	create a multicast socket by address
<code>Multicast host port</code>	create a multicast socket by host and port
<code>Multicast addr port</code>	create a multicast socket by address and port

Derivation summary

Derived from	Description
<code>UdpSocket</code>	the UDP socket class

Mail [aleph:net]

Description

The `Mail` class is a mail delivery object which manages to contact an MTA *Mail Transport Agent* in order to deliver a message to one or several recipients. By default, the object contacts the local MTA, but this behavior can be changed with the `set-mta-address` method. The class implements the recipient address syntax as specified by RFC822. At construction, the instance is empty. Only the recipient address needs to be specified. The `send` method send the message by contacting the MTA. If an error occurs, an exception is raised.

Constructors Summary

Constructor	Description
Mail	create an empty message

Methods Summary

Method	Description
<code>to</code>	add a new recipient in the to list
<code>cc</code>	add a new recipient in the cc list
<code>bcc</code>	add a new recipient in the bcc list
<code>add</code>	add literals to the message buffer
<code>addln</code>	add literals followed by a newline
<code>send</code>	send the message by contacting the MTA
<code>subject</code>	set the message subject
<code>get-mta-address</code>	get the mta IP address
<code>set-mta-address</code>	set the mta IP address
<code>get-mta-port</code>	get the mta IP port number
<code>set-mta-port</code>	set the mta IP port number

Mail:to

- return: none
- arguments: String

The `to` method adds one or several address to the *destination* list. The address format must conform to RFC822. Multiple address are coma separated. Multiple call to this method is possible.

Mail:cc

- return: none
- arguments: String

The **cc** method adds one or several address to the *copy* list. The address format must conform to RFC822. Multiple address are coma separated. Multiple call to this method is possible.

Mail:bcc

- return: none
- arguments: String

The **bcc** method adds one or several address to the *blind copy* list. The address format must conform to RFC822. Multiple address are coma separated. Multiple call to this method is possible. The *blind copy* list is not included in the message header.

Mail:add

- return: none
- arguments: String ...

The **add** method adds one or several literals to the message buffer. This is the normal way to fill a message buffer by string line.

Mail:addln

- return: none
- arguments: String ...

The **addln** method adds one or several literals to the message buffer. A newline character is added at the end of the line. This is a similar way to fill a message buffer by string line.

Mail:send

- return: none
- arguments: none

The **send** method request a message delivery by contacting the MTA. Once the MTA has been contacted, the message header and the message body is transfered. The MTA is responsible to deliver the message to the appropriate recipients.

Mail:subject

- return: none
- arguments: String

The **subject** method sets the message subject string line.

Mail:set-mta-address

- return: none
- arguments: String

The **set-mta-address** method sets the MTA IP address that the class needs to contact for mail request. The address can be an fully qualified host name or an IP number.

Mail:get-mta-address

- return: String
- arguments: none

The **get-mta-address** method returns the current MTA IP address for this mail object.

Mail:set-mta-port

- return: none
- arguments: Integer

The **set-mta-port** method set the current MTA IP port number for this mail object. With the MTA IP address, the MTA to contact for mail request is uniquely defined. The default port value is 25.

Mail:get-mta-port

- return: Integer
- arguments: none

The **get-mta-port** method returns the current MTA IP port number for this mail object. The default port value is 25.

APPENDIX H

Networking Functions

This chapter is a reference of the Aleph networking functions. The functions described here are part of the **aleph-net** library. The library must be loaded prior any use of these functions. Once the library is loaded, all functions are located in the **aleph:net** nameset.

Table 13 Aleph network call functions

Function	Description
get-loopback	return the loopback name
get-tcp-service	return the tcp service name by id
get-udp-service	return the udp service name by id

aleph:net:get-loopback

- return: String
- arguments: none

The **get-loopback** function returns the name of the machine loopback. On a UNIX system, that name is `localhost`.

aleph:net:get-tcp-service

- return: String
- arguments: Integer

The **get-tcp-service** function returns the name of the tcp service given its port number. For example, the tcp service at port 13 is the *daytime* server.

aleph:net:get-udp-service

- return: String
- arguments: Integer

The **get-udp-service** function returns the name of the udp service given its port number. For example, the udp service at port 19 is the *chargen* server.

APPENDIX I

WWW/CGI Classes and Functions

This appendix is a reference of the **Aleph** web services. The classes described here are part of the **aleph-www** library. The library must be loaded prior any use of these classes. Once the library is loaded, all classes are located in the **aleph:www** namespaces.

Table 14 Aleph web classes

Object	Description
Url	url class
Cookie	http cookie class
CgiQuery	cgi query class
HtmlPage	html page class
XHtmlPage	xhtml page class

For each class, a predicate is provided.

Table 15 Aleph web class predicates

Object	Predicate
Url	url-p
Cookie	cookie-p
CgiQuery	cgi-query-p
HtmlPage	html-page-p

Url [aleph:www]

Description

The `Url` class is the `Universal Resource Locator` manipulation class. The class can be used to either parse a *URL* or build one by pieces. The class do automatically the escape sequence conversion.

Derivation summary

Derived from	Description
Object	the base object class

Constructors Summary

Constructor	Description
<code>Url</code>	create an empty url
<code>Url name</code>	create a URL by name

Methods Summary

Method	Description
<code>parse</code>	parse a string
<code>get-scheme</code>	returns the url scheme
<code>get-host</code>	returns the url host
<code>get-port</code>	returns the url port
<code>get-path</code>	returns the url path
<code>get-query</code>	returns the url query
<code>get-fragment</code>	returns the url fragment

Url:parse

- return: `none`
- arguments: `String`

The **parse** method reset the URL object, parse the string argument and fill the URL object with the result.

Url:get-scheme

- return: String
- arguments: none

The **get-scheme** method returns the scheme of the parsed *URL* object. The default scheme is `http` if not specified at object construction.

Url:get-host

- return: String
- arguments: none

The **get-host** method returns the host of the parsed url.

Url:get-port

- return: Integer
- arguments: none

The **get-port** method returns the host of the parsed url. The default port is 80 if not specified. With some scheme, the port value do not make sense.

Url:get-path

- return: String
- arguments: none

The **get-path** method returns the path of the parsed url. The default path is `'/'` if not specified.

Url:get-query

- return: String
- arguments: none

The **get-query** method returns the complete query string of the parsed *URL*.

Url:get-fragment

- return: String
- arguments: none

The **get-fragment** method returns the complete query string of the parsed *URL*.

CgiQuery [aleph:www]

Description

The `CgiQuery` class is a special object that parse a CGI query string and provides methods to access form values by key. The class takes care of converting the escaped characters.

Derivation summary

Derived from	Description
Object	the base object class

Constructors Summary

Constructor	Description
<code>CgiQuery</code>	create an empty query
<code>CgiQuery query</code>	create a query by value

Methods Summary

Method	Description
<code>get</code>	get a value by key
<code>parse</code>	parse the query string
<code>exists-p</code>	check if a key exist
<code>length</code>	return the number of keys
<code>lookup</code>	return the key index by name
<code>get-name</code>	get the key name by index
<code>get-value</code>	get a key value by index
<code>get-query</code>	return the query string

CgiQuery:get

- return: `String`
- arguments: `String`

The `get` method returns the value associated with the key specified by the argument. If the key does not exist, the empty string is returned.

CgiQuery:parse

- return: none
- arguments: String

The **parse** method reset the query object, parse the string argument and fill the query object with the result.

CgiQuery:length

- return: Integer
- arguments: none

The **length** method returns the number of keys available in the query object.

CgiQuery:exists-p

- return: Boolean
- arguments: String

The **exists-p** predicate returns true if the key argument exists in this query object.

CgiQuery:lookup

- return: Integer
- arguments: String

The **lookup** method returns index of the key argument in the query object. If the key does not exist, -1 is returned.

CgiQuery:get-name

- return: String
- arguments: Integer

The **get-name** method returns the name of the key specified by the index argument.

CgiQuery:get-value

- return: String
- arguments: Integer

The **get-value** method returns the value associated with the key specified by the index argument.

CgiQuery:get-query

- return: String
- arguments: none

The **get-query** method returns the original query string used during the parsing.

HtmlPage [aleph:www]

Description

The **HtmlPage** class is the primary interface to generate *HTML* code. The class operates by filling the header and the body of the page with HTML statements. Several methods are provided to ease the task of the page generation. The HTML version is assumed to be *strict 4.01*. Since HTML 4.01 is designed to work with *style sheet*, the user must be prepared to handle this when generating its own HTML page. The **HtmlPage** constructor takes no argument. The basic method used to add something in the header is the `add-head` method which take one or several literal arguments. The `add-title` method adds a title to the header. The `add-style` adds the style sheet definition to the header. The `add-author` add the author's name to the page header. Finally, the `add-meta` method adds a *meta* statement to the header in the form of name and content. The `add-body` method adds any literal object into the body buffer.

Derivation summary

Derived from	Description
Object	the base object class

Constructors Summary

Constructor	Description
HtmlPage	create an empty html page

Methods Summary

HtmlPage:add-http

- return: none
- arguments: [Literal...]

The **add-http** method adds one or more literal objects into the http buffer. This methods can be used to add specific http attributes.

HtmlPage:add-head

- return: none
- arguments: [Literal...]

The **add-head** method adds one or more literal objects into the head buffer.

Method	Description
add-http	add http content in the http buffer
add-head	add literals in the head buffer
add-body	add literals in the body buffer
add-meta	add a meta in the head buffer
add-title	add a title in the head buffer
add-style	add a style sheet in the head buffer
add-author	add the author in the head buffer
add-cookie	add a cookie in the http buffer
get-buffer	get the full page in a buffer
write-cgi	write the page with cgi header
write-http	write the page http buffer
write-head	write the page head buffer
write-body	write the page head buffer
write-page	write the page

HtmlPage:add-body

- return: none
- arguments: [Literal...]

The **add-head** method adds one or more literal objects into the body buffer.

HtmlPage:add-meta

- return: none
- arguments: String String

The **add-meta** adds a meta mark-up in the head buffer. The first argument is mark-up name and the second argument is the mark-up value.

HtmlPage:add-title

- return: none
- arguments: String

The **add-title** adds a title mark-up in the head buffer. The title is the string argument.

HtmlPage:add-author

- return: none
- arguments: String

The **add-author** adds an author mark-up in the head buffer. The author is the string argument.

HtmlPage:add-style

- return: none
- arguments: String

The **add-style** adds a style sheet name in the head buffer. Several style sheets can be specified. The style sheet path is specified as a string argument.

HtmlPage:add-cookie

- return: none
- arguments: Cookie

The **add-cookie** adds a cookie content in the http buffer. The cookie object is translated into a string http content value and added into the http buffer.

HtmlPage:get-buffer

- return: Buffer
- arguments: none

The **get-buffer** method returns the full content of the html page into a buffer object. This method is useful for one who want to query the page length and then write it on an output stream. Remember that the buffer class as a write method to do so.

HtmlPage:write-http

- return: none
- arguments: Output

The **write-http** method writes the http buffer into the output stream specified as the argument.

HtmlPage:write-head

- return: none
- arguments: Output

The **write-head** method writes the head buffer into the output stream specified as the argument.

HtmlPage:write-body

- return: none
- arguments: Output

The **write-body** method writes the body buffer into the output stream specified as the argument.

HtmlPage:write-page

- return: none
- arguments: Output

The **write-page** method writes the whole page into the output stream specified as the argument.

HtmlPage:write-cgi

- return: none
- arguments: Output

The **write-cgi** method writes first a CGI server reply in the form of content type and status and then the whole page into the output stream specified as the argument.

XHtmlPage [aleph:www]

Description

The **XHtmlPage** class is specialized class that produces *XHTML* code. The class is derived from the **HtmlPage** class and inherits all methods.

Derivation summary

Derived from	Description
HtmlPage	the HTML page class

Constructors Summary

Constructor	Description
XHtmlPage	create an empty xhtml page

Methods Summary

Method	Description
get-language	get the xml language
set-language	set the xml language

XHtmlPage:set-language

- return: none
- arguments: String

The **set-language** method set the xml language tag in the html document. By default, the xml language is set to "en".

XHtmlPage:get-language

- return: String
- arguments: none

The **get-language** method returns the xml language tag in the html document. By default, the xml language is set to "en".

Cookie [aleph:www]

Description

The **Cookie** class is a special class designed to handle cookie setting within a CGI script. A cookie is name/value pair that is set by the server and stored by the HTTP client. Further connection with the client will result with the cookie value transmitted by the client to the server. A cookie has various parameters that controls its existence and behavior. The most important one the *cookie maximum age* that is defined in second. A null value tells the client to discard the cookie. A cookie without maximum age is valid only during the HTTP client session. A cookie can be added to the HTML page with the `set-cookie` method. A cookie can be constructed with a name/value pair. An optional third argument is the maximum age.

Derivation summary

Derived from	Description
Object	the base object class

Constructors Summary

Constructor	Description
<i>Cookie name value</i>	create a new cookie with name and value
<i>Cookie name value age</i>	create a new cookie with name, value and age

Methods Summary

Cookie:get-name

- return: String
- arguments: none

The **get-name** method returns the cookie name. This is the name store on the HTTP client.

Cookie:set-name

- return: none
- arguments: String

The **set-name** method sets the cookie name. This is the name store on the HTTP client.

Cookie:get-value

Method	Description
get-name	return the cookie name
set-name	set the cookie name
get-value	return the cookie value
set-value	set the cookie value
get-max-age	return the cookie maximum age
set-max-age	set the cookie maximum age
get-path	return the cookie path
set-path	set the cookie path
get-domain	return the cookie domain
set-domain	set the cookie domain
get-comment	return the cookie comment
set-comment	set the cookie comment
get-secure	return the cookie secure flag
set-secure	set the cookie secure flag
to-string	return an HTTP cookie string value

■ return: String

■ arguments: none

The **get-value** method returns the cookie value. This is the value stored on the HTTP client bounded by the cookie name.

Cookie:set-value

■ return: none

■ arguments: String

The **set-value** method sets the cookie value. This is the value store on the HTTP client bounded by the cookie name.

Cookie:get-maximum-age

■ return: Integer

■ arguments: none

The **get-maximum-age** method returns the cookie maximum age. The default value is -1, that is, no maximum age is set and the cookie is valid only for the HTTP client session.

Cookie:set-maximum-age

■ return: none

■ arguments: Integer

The **set-maximum-age** method sets the cookie maximum age. A negative value is reset to -1. A 0 value tells the HTTP client to discard the cookie. A positive value tells the HTTP client to store the cookie for the remaining seconds.

Cookie:get-path

■ return: String

■ arguments: none

The **get-path** method returns the cookie path value. The path determines for which HTTP request the cookie is valid.

Cookie:set-path

■ return: none

■ arguments: String

The **set-path** method sets the cookie path value. The path determines for which HTTP request the cookie is valid.

Cookie:get-domain

■ return: String

■ arguments: none

The **get-domain** method returns the cookie domain value.

Cookie:set-path

■ return: none

■ arguments: String

The **set-domain** method sets the cookie domain value. It is string recommended to use the originator domain name since many HTTP client can reject cookie those domain name does not match the originator name.

Cookie:get-comment

■ return: String

■ arguments: none

The **get-comment** method returns the cookie comment value.

Cookie:set-comment

■ return: none

■ arguments: String

The **set-comment** method sets the cookie comment value.

Cookie:get-secure

■ return: String

■ arguments: none

The **get-secure** method returns the cookie secure flag.

Cookie:set-comment

■ return: none

■ arguments: String

The **set-secure** method sets the cookie secure flag. When a cookie is secured, it is only returned by the HTTP client if a connection has been secured (i.e use HTTPS).

Cookie:to-string

■ return: String

■ arguments: none

The **to-string** method returns a string formatted as an HTTP Set-cookie request. Normally this method should not be called since the **HtmlPage** `add-cookie` method takes care of such thing.

BIBLIOGRAPHY

- [1] *RFC 738 - Time server*, 1977.
- [2] *RFC 791 - DARPA Internet Program Protocol Specification*, 1981.
- [3] Revised Report on the Algorithmic Language Scheme. Technical report, November 1991.
- [4] *C++ Language Reference Manual*, 1996.
- [5] *RFC 2101 - IPV4 Address Behaviour Today*, 1997.
- [6] *RFC 2373 - IP Version 6 Addressing Architecture*, 1998.
- [7] Guy L. Steele Jr. *Common Lisp, The Language*. 1990.
- [8] Donald E. Knuth. *The Art of Computer Programming, Volume 1*. 1997.
- [9] Donald E. Knuth. *The Art of Computer Programming, Volume 2*. 1997.
- [10] Donald E. Knuth. *The Art of Computer Programming, Volume 3*. 1997.
- [11] George Springer and Daniel P. Friedman. *Scheme and the Art of Programming*. 1997.
- [12] W. Richard Stevens. *TCP/IP Illustrated Volume 1*. 1994.
- [13] W. Richard Stevens. *UNIX Network Programming - Interprocess Communication*. 1998.
- [14] W. Richard Stevens. *UNIX Network Programming - Socket API*. 1998.
- [15] Bjarne Stroustrup. *The C++ Programming Language*. 2000.

INDEX

- absolute-path
 - aleph:sio function, 69
- accept
 - TcpSocket method, 101
 - UdpSocket method, 109
- add
 - Mail method, 118
 - PrintTable method, 83
 - Selector method, 67
 - Time method, 74
- add-author
 - HtmlPage method, 130
- add-body
 - HtmlPage method, 130
- add-cookie
 - HtmlPage method, 131
- add-head
 - HtmlPage method, 129
- add-http
 - HtmlPage method, 129
- add-meta
 - HtmlPage method, 130
- add-style
 - HtmlPage method, 130
- add-title
 - HtmlPage method, 130
- addln
 - Mail method, 118
- Address
 - aleph:net:Address class, 19
 - constructors summary, 93
 - get-canonical-name method, 20
 - get-ip-address method, 20
 - methods summary, 93
 - object reference, 93
- aleph:sys:get-host-name, 20
- AOD
 - Aleph Object Database, 39
 - structured elements, 39
- bcc
 - Mail method, 118
- bind
 - Socket method, 97
- BROADCAST
 - Socket constant, 95
- cc
 - Mail method, 117
- cell
 - definition, 39
- CGI
 - Web service, 33
- CgiQuery
 - constructors summary, 127
 - derivation summary, 127
 - methods summary, 127
 - object reference, 127
- close
 - InputFile method, 47
 - OutputFile method, 57
 - Socket method, 98
- collection
 - definition, 40
- command
 - create, 41
 - import, 41
 - open, 41
 - save, 41
- compute
 - Digest method, 87
- connect
 - Socket method, 97
- Cookie
 - constructors summary, 135
 - derivation summary, 135
 - methods summary, 135
 - object reference, 135
- create
 - command usage, 41
- data
 - with cell, 39
- Datagram
 - derivation summary, 107
 - object reference, 107
- Digest
 - constructors summary, 87
 - derivation summary, 87
 - methods summary, 87
 - object reference, 87
- dir-p
 - aleph:sio function, 69
- Directory, 6
 - constructors summary, 65
 - methods summary, 65
 - object reference, 65
- DNS, 19
- DONT-ROUTE
 - Socket constant, 95
- eof-p
 - Input method, 46
 - Socket method, 99
- errorln
 - Output method, 55
- ErrorTerm, 5
- exists-p

- CgiQuery method, 128
- exit
 - aleph:sys function, 79
- file-p
 - aleph:sio function, 69
- flush
 - OutputString method, 59
- format
 - PrintTable method, 84
- format-date
 - Time method, 76
- format-time
 - Time method, 76
- get
 - CgiQuery method, 127
 - InputString method, 51
 - PrintTable method, 83
- get-buffer
 - HtmlPage method, 131
- get-buffer-length
 - Input method, 46
- get-canonical-name
 - Address method, 94
- get-column-direction
 - PrintTable method, 85
- get-column-fill
 - PrintTable method, 85
- get-column-size
 - PrintTable method, 85
- get-columns
 - PrintTable method, 84
- get-comment
 - Cookie method, 137
- get-day-of-month
 - Time method, 75
- get-day-of-week
 - Time method, 75
- get-day-of-year
 - Time method, 75
- get-domain
 - Cookie method, 137
- get-env
 - aleph:sys function, 80
- get-files
 - Directory method, 66
- get-fragment
 - Url method, 126
- get-host
 - Url method, 126
- get-host-name
 - aleph:sys function, 80
- get-hours
 - Time method, 74
- get-ip-address
 - Address method, 93
- get-ip-vector
 - Address method, 93
- get-language
 - XHtmlPage method, 133
- get-list
 - Directory method, 66
- get-loopback
 - aleph:net function, 121
- get-maximum-age
 - Cookie method, 136
- get-minutes
 - Time method, 74
- get-month
 - Time method, 74
- get-mta-address
 - Mail method, 118
- get-mta-port
 - Mail method, 119
- get-name
 - Address method, 93
 - CgiQuery method, 128
 - Cookie method, 135
 - Directory method, 66
 - InputFile method, 47
 - InputMapped method, 50
 - OutputFile method, 57
- get-offset
 - InputMapped method, 50
- get-path
 - Cookie method, 136
 - Url method, 126
- get-peer-address, 24
 - Socket method, 99
- get-peer-port, 24
 - Socket method, 99
- get-pid
 - aleph:sys function, 80
- get-port
 - Url method, 126
- get-primary
 - Terminal method, 64
- get-query
 - CgiQuery method, 128
 - Url method, 126
- get-rows
 - PrintTable method, 84
- get-scheme
 - Url method, 125
- get-secondary
 - Terminal method, 64

- get-seconds
 - Time method, 74
- get-secure
 - Cookie method, 137
- get-socket-address, 24
 - Socket method, 99
- get-socket-port, 24
 - Socket method, 99
- get-subdirs
 - Directory method, 66
- get-tcp-service, 21
 - aleph:net function, 121
- get-time
 - Time method, 74
- get-udp-service, 21
 - aleph:net function, 121
- get-user-name
 - aleph:sys function, 80
- get-utc-day-of-month
 - Time method, 76
- get-utc-day-of-week
 - Time method, 76
- get-utc-day-of-year
 - Time method, 76
- get-utc-hours
 - Time method, 75
- get-utc-minutes
 - Time method, 75
- get-utc-month
 - Time method, 75
- get-utc-seconds
 - Time method, 75
- get-utc-year
 - Time method, 76
- get-value
 - CgiQuery method, 128
 - Cookie method, 135
- get-year
 - Time method, 74
- HOP-LIMIT
 - Socket constant, 96
- HtmlPage
 - constructors summary, 129
 - derivation summary, 129
 - methods summary, 129
 - object reference, 129
- import
 - command usage, 42
- Input
 - object reference, 45
- input-get
 - Selector method, 68
- input-length
 - Selector method, 68
- InputFile
 - constructors summary, 47
 - derivation summary, 47
 - methods summary, 45, 47
 - object reference, 47
- InputMapped
 - constructors summary, 49
 - derivation summary, 49
 - methods summary, 49
 - object reference, 49
- InputString
 - constructors summary, 51
 - derivation summary, 51
 - methods summary, 51
 - object reference, 51
- InputTerm, 5
 - constructors summary, 53
 - derivation summary, 53
 - methods summary, 53
 - object reference, 53
- interpreter
 - system information, 9
- IP address, 19
- IPv4
 - address example, 19
- IPv6
 - address example, 19
- ipv6-p
 - Socket method, 97
- KEEP-ALIVE
 - Socket constant, 96
- length
 - CgiQuery method, 128
 - InputFile method, 48
 - InputMapped method, 49
- LINGER
 - Socket constant, 96
- listen
 - TcpSocket method, 101
- lookup
 - CgiQuery method, 128
- lseek
 - InputFile method, 48
 - InputMapped method, 49
- Mail
 - constructors summary, 117
 - methods summary, 117
 - object reference, 117
- MAX-SEGMENT-SIZE

- Socket constant, 96
- mkdir
 - Directory method, 65
- Multicast
 - constructors summary, 115
 - derivation summary, 115
 - object reference, 115
- MULTICAST-HOP-LIMIT
 - Socket constant, 96
- MULTICAST-LOOPBACK
 - Socket constant, 96
- newline
 - Output method, 55
 - Socket method, 98
- NO-DELAY
 - Socket constant, 97
- ODB
 - Object database library, 39
- open
 - command usage, 41
- output-get
 - Selector method, 68
- output-length
 - Selector method, 68
- OutputFile
 - constructors summary, 57
 - derivation summary, 57, 59
 - methods summary, 55, 57
 - object reference, 55, 57
- OutputString
 - constructors summary, 59
 - methods summary, 59
 - object reference, 59
- OutputTerm, 5
 - constructors summary, 61
 - derivation summary, 61
 - object reference, 61
- parse
 - CgiQuery method, 127
 - Url method, 125
- peer
 - address and port, 21
- PrintTable
 - constructors summary, 83
 - derivation summary, 83
 - methods summary, 83
 - object reference, 83
- pushback
 - Input method, 46
 - Socket method, 99
- random
 - aleph:sys function, 79
- RCV-SIZE
 - Socket constant, 96
- read
 - Input method, 45
 - Socket method, 97, 98
- readln
 - Input method, 45
 - Socket method, 98
- record
 - definition, 39
- relative-path
 - aleph:sio function, 70
- REUSE-ADDRESS
 - Socket constant, 95
- RFC 738, 23
- rmdir
 - aleph:sio function, 70
 - Directory method, 65
- rmfile
 - aleph:sio function, 70
 - Directory method, 65
- save
 - command usage, 42
- Selector
 - constructors summary, 67
 - methods summary, 67
 - object reference, 67
- send
 - Mail method, 118
- set
 - InputString method, 51
 - PrintTable method, 84
- set-column-direction
 - PrintTable method, 85
- set-column-fill
 - PrintTable method, 85
- set-column-size
 - PrintTable method, 84
- set-comment
 - Cookie method, 137
- set-eof-character
 - InputTerm method, 53
- set-eof-ignore
 - InputTerm method, 53
- set-language
 - XHtmlPage method, 133
- set-maximum-age
 - Cookie method, 136
- set-mta-address
 - Mail method, 118
- set-mta-port
 - Mail method, 119

- set-name
 - Cookie method, 135
- set-option
 - Socket method, 99
- set-path
 - Cookie method, 137
- set-primary
 - Terminal method, 63
- set-secondary
 - Terminal method, 63
- set-value
 - Cookie method, 136
- shutdown
 - Socket method, 97
- sleep
 - aleph:sys function, 79
- SND-SIZE
 - Socket constant, 96
- Socket
 - constant summary, 95
 - derivation summary, 95
 - methods summary, 95
 - object reference, 95
- socket-p, 25
- sort
 - aleph:txt function, 89
- subject
 - Mail method, 118
- table
 - definition, 40
- tcp-client-p, 25
- TcpClient
 - constructors summary, 103
 - derivation summary, 103
 - object reference, 103
- TcpServer
 - constructors summary, 105
 - derivation summary, 105
 - object reference, 105
- TcpSocket
 - constructors summary, 101
 - derivation summary, 101
 - methods summary, 101
 - object reference, 101
- Terminal, 5
 - constructors summary, 63
 - derivation summary, 63
 - methods summary, 63
 - object reference, 63
- Time
 - constructors summary, 73
 - methods summary, 73
 - object reference, 73
- to
 - Mail method, 117
- to-string
 - Cookie method, 138
 - OutputString method, 59
- udp-client-p, 25
- UdpClient
 - constructors summary, 111
 - derivation summary, 111
 - object reference, 111
- UdpServer
 - constructors summary, 113
 - derivation summary, 113
 - object reference, 113
- UdpSocket
 - constructors summary, 109
 - derivation summary, 109
 - methods summary, 109
 - object reference, 109
- URL
 - class description, 33
- Url
 - constructors summary, 125
 - derivation summary, 125
 - methods summary, 125
 - object reference, 125
- utc-format-cookie
 - Time method, 77
- utc-format-date
 - Time method, 77
- utc-format-rfc
 - Time method, 77
- utc-format-time
 - Time method, 77
- valid-p
 - Input method, 45
 - Socket method, 98
- wait
 - Selector method, 67
- wait-all
 - Selector method, 68
- write
 - Output method, 55
 - Socket method, 98
- write-body
 - HtmlPage method, 131
- write-cgi
 - HtmlPage method, 131
- write-head
 - HtmlPage method, 131
- write-http

- HtmlPage method, 131
- write-page
 - HtmlPage method, 131
- writeln
 - Output method, 55
 - Socket method, 98
- XHtmlPage
 - constructors summary, 133
 - derivation summary, 133
 - methods summary, 133
 - object reference, 133

Colophon

This manual was written for the \LaTeX documentation preparation system. A custom document class was designed by the author. The document style has been simplified as to produce a high quality technical manual. Title, chapter and section names have been produced with an Helvetica font. The document has been produced with a 10 points Times font. Both fonts are assumed to be in the public domain. The documentation is available in both A4 and letter format.