Telescript Language Reference

October 1995

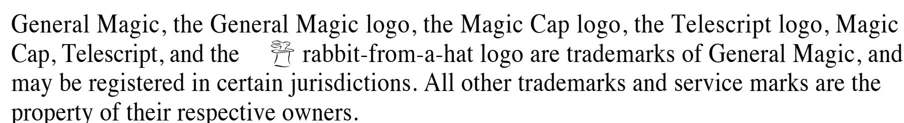
General Magic, Inc.
420 North Mary Avenue
Sunnyvale, CA 94086

© 1991 - 1995 General Magic, Inc.
All rights reserved

Copyright and Trademark

The general idea of a remote programming language is in the public domain. Anyone is free to formulate his or her own expression of this idea by devising a unique language structure, syntax and vocabulary. However, General Magic exercised a great deal of original expression when choosing the structure, syntax and vocabulary of the Telescript language. Therefore, General Magic believes that it owns the copyright in the structure, syntax and vocabulary of the Telescript programming language. General Magic believes that no one else can create a Telescript language interpreter without violating its copyright. General Magic is also seeking patent protection on some of the novel inventions in the Telescript architecture and software agent functionality. Finally, General Magic owns the trademark "Telescript," which has been registered in the U.S. and many other jurisdictions. General Magic uses the "Telescript" trademark to identify General Magic's Telescript software.

General Magic wants to promote the use of the Telescript language as an enabling technology for creating "smart" networks. Therefore, we have an open, non-discriminatory policy on licensing our Telescript software for research and development, and commercial purposes. We also encourage you to write programs in the Telescript language. Feel free to incorporate any of the sample programs contained in this document into your programs.



General Magic, the General Magic logo, the Magic Cap logo, the Telescript logo, Magic Cap, Telescript, and the rabbit-from-a-hat logo are trademarks of General Magic, and may be registered in certain jurisdictions. All other trademarks and service marks are the property of their respective owners.

Limit of Liability/Disclaimer of Warranty

THIS BOOK IS PROVIDED TO YOU "AS IS." Even though General Magic has reviewed this book in detail, GENERAL MAGIC MAKES NO REPRESENTATION OR WARRANTIES, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THE ACCURACY OR COMPLETENESS OF THE CONTENTS OF THIS BOOK. GENERAL MAGIC SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OR MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE AND SHALL IN NO EVENT BE LIABLE FOR ANY LOSS OF PROFIT OR ANY OTHER COMMERCIAL DAMAGE, INCLUDING BUT NOT LIMITED TO SPECIAL, INCIDENTAL, CONSEQUENTIAL, OR OTHER DAMAGES, EVEN IF MAGIC KNOWS OR SHOULD KNOW OF THE POSSIBILITY OF SUCH DAMAGES . Some states do not allow for the exclusion or limitation of implied warranties or incidental or consequential damage. So, the exclusions in this paragraph might not apply to you. This warranty gives you specific legal rights. You may also have other rights which vary from state to state.

Important to Someone

Restricted Rights. For defense agencies: Use, duplication, or disclosure is subject to the restrictions set forth in subparagraph (c)(1)(ii) of DFAR section 252.227-7013 and its successors. For civilian agencies: Use, duplication, or disclosure is subject to the restrictions set forth in subparagraphs (a) through (d) of FAR section 52.227-19 and its successors. Unpublished—rights reserved under the copyright laws of the United States.

General Magic, Inc.
420 North Mary Avenue
Sunnyvale, CA 94086 USA

Tel.: 408 774 4000
Fax: 408 774 4010
E-mail: dev-info@genmagic.com
URL: http://www.genmagic.com/

Patent Pending

Portions of the Magic Cap software and the Telescript software are patent pending in the United States and other countries.

Table of Contents

Preface xi

Introduction 1

About the technology 1

Telescript model 1
Telescript language 2
Telescript engine 3
Telesphere 4

About this manual 4

Audience 4
Informal conventions 5
Formal conventions 5
References 5

Part One—Language Concepts 7

Object composition 8

Operations 8

The operation concept 8
Defining an operation's interface 8
Defining an operation's implementation 9
Implementing a method with a block 9
Maintaining local variables 10

Attributes 10

The attribute concept 10
Defining an attribute's interface 10
Defining an attribute's implementation 11
Maintaining properties 11

Constraints 11

The constraint concept 11
Defining a constraint's type 12
Defining a constraint's passage 12
Specifying a class 12
Searching for a class 13

Object manipulation 14

Object references 14

The reference concept 14
Acquiring a protected reference 14
Acquiring a voided reference 14
Protecting objects 15

Object manipulation 15

Copying an object 15
Destroying an object 16

Object ownership 16

The ownership concept 16
Isolating an object 17
Locking an object 17
Freezing an object 17
Thawing an object 18
Transferring ownership 18

Object aggregation 19

Class definitions 19

The class concept 19
The class family concept 19

Specifying a class's interface 19
Specifying a class's implementation 20
Elaborating upon inheritance 20
Class relationships 21
Relating one flavor to another 21
Relating one mix-in to another 21
Relating one class to another 22
Ordering a class and its superclasses 22
Searching a class and its superclasses 22
The constructor 23
The constructor concept 23
Deciding the constructor's formal arguments 23
Deciding the constructor's actual arguments 23
Performing the constructor 24

Object terminology 25

Part Two—Language 27

Basic constructs 28
Statements and expressions 28
Operation and cascade requests 28
Objects and their identifiers 28
Global variables 29

Definitions 30
Module definitions 30
Interface definitions 30
Class definitions 31
Defining a class 31
Defining a class's formal parameters 31
Defining a class's immediate superclasses 32
Defining a class's features 32
Defining features' requesters 32
Defining features' responders 33
Defining attributes or properties 33
Defining operations 33
Sealing features 33
Attribute definitions 34
Defining an attribute 34
Defining an attribute's signature 34
Defining an attribute's getter or setter 35
Operation definitions 35
Defining an operation 35
Defining an operation's signature 35
Defining an operation's expected arguments 36
Defining an operation's named arguments 36
Defining an operation's unnamed arguments 37
Defining an operation's method 37
Defining a block 38
Constraint definitions 38
Defining a constraint 38
Defining a constraint's type 38
Defining a constraint's passage 39
Defining a class specifier 39

Statements 40
Basic statements 40
Using an expression as a statement 40
The do statement 40
The if statement 40

The if-else statement 41
The return statement 41
Iterative statements.......... 41
The loop statement 41
The while statement 42
The repeat statement 42
The for-to statement 42
The for-in statement 42
The continue statement 43
The break statement 43
Exception statements.......... 43
Declaring catchphrases 43
The try statement 44
The throw statement 44
Process statements.......... 44
The own statement 44
The restrict statement 44
The use statement 45

Expressions.......... 46
Basic expressions.......... 46
Accessing an object 46
Assigning an object 47
Asserting an object's type 47
Operator expressions.......... 47
Applying a prefix operator 48
Applying an infix operator 48
General operation expressions.......... 49
Specifying the responder 49
Specifying the arguments 50
Requesting a cascade of operations 50
Requesting an operation 51
Requesting a setter 51
Special operation expressions.......... 51
Requesting the new operation 52
Requesting a get operation 52
Requesting a set operation 52
Requesting a getter 53
Escalating the current operation 53
Literal expressions.......... 53
Denoting a bit 54
Denoting a bit string 54
Denoting a boolean 54
Denoting a character 54
Denoting an identifier 54
Denoting an integer 54
Denoting nil 55
Denoting an octet 55
Denoting an octet string 55
Denoting a real 55
Denoting a string 55
Escape expressions.......... 56

Programs.......... 57
Abstract programs.......... 57
Source programs.......... 57
Including a preprocessing directive 57
Including a break 57
Including a comment 58
Including an escape sequence 58
Including a reserved word 59
Including a character category 59
Including a character 60

Object programs.. 62

Part Three—Predefined Class Concepts.......................... 63

Places.. 64
Organizing the telesphere.. 64
Organizing a region 64
Organizing an engine 64
Addressing a place.. 65
Specifying a region 65
Specifying a place 65
Specifying routing advice 65
Managing occupants... 65
Entering a place 65
Exiting a place 66
Keeping track of occupants 66

Agents 67
Traveling to other places... 67
Constructing a ticket 67
Satisfying a ticket 68
Receiving a ticket stub 68
Going to another place 68
Sending clones to other places 68
Selecting a route.. 69
Ensuring a route 69
Using reservable means 69
Using existing connection means 69
Meeting other agents... 70
Constructing a petition 70
Satisfying a petition 70
Managing meetings... 71
Beginning a meeting 71
Ending a meeting 71
Keeping track of acquaintances 71

Processes.. 72
Defining a process... 72
Branding a process 72
Phasing a process 72
Activating a process 72
Prioritizing a process 73
Terminating a process 73
Naming a process.. 73
Specifying an authority 74
Specifying a process 74
Contacting a process through an operation.. 74
Contacting a process through a package.. 74
Offering packages 74
Searching packages for objects 75
Contacting a process through an event.. 75
Categorizing an event 75
Sending a signal 76
Enabling or disabling a signal 76
Receiving a signal 76
Contacting a process through a resource... 76
Using a resource 77
Using a resource exclusively 77
Using a resource conditionally 77
Losing contact with a process.. 77

Permits 78
Defining a permit 78
Granting an action 78
Granting a resource 78
Granting a form of recognition 79
Receiving a permit 79
Receiving a native permit 79
Receiving a regional permit 80
Receiving a local permit 80
Receiving a temporary permit 80
Reconciling permits 81
Intersecting two permits 81
Ordering two permits 81
Ordering two capabilities 81
Increasing or decreasing a capability 81
Enforcing the current permit 81
Determining the current permit 82
Violating the current permit 82
Exhausting the current permit 82

Patterns 83
Defining and using a pattern 83
Defining a pattern 83
Using a pattern 83
Requiring a match 83
Requiring a match 84
Requiring an anchored match 84
Requiring successive matches 84
Requiring a repeated match 84
Requiring a single match 84
Requiring a character 85
Requiring a character with certain attributes 85
Requiring a character in a certain list 85
Requiring a character not in a certain list 86
Requiring a character in a certain interval 86
Requiring a character with a certain name 86
Requiring a character 86

Calendar times 87
Defining a calendar time 87
Accessing a calendar time 87
Accessing the time 87
Accessing the date 87
Normalizing a calendar time 88
Normalizing the time 88
Normalizing the date 88

Part Four—Predefined Classes 91
Legend 92

Agent 93
Authenticator 95
Bit 96
Bit String 97
Boolean 98
Calendar Time 99
Cased 102
Character 103
Class 105

Class Exception 107
Class Name 108
Collection 109
Collection Exception 112
Compared 113
Death Event 114
Dictionary 115
Engine Place 118
Equal 120
Event 121
Event Process 122
Exception 124
Execution Exception 125
Existing Connection Means 127
Exit Event 128
Identifier 129
Integer 130
Iterator 132
Kernel Exception 133
List 135
Means 139
Meeting Agent 140
Meeting Exception 141
Meeting Place 142
Miscellaneous Exception 144
Named 145
Nil 146
Number 147
Object 150
Octet 154
Octet String 155
Ordered 157
Package 158
Package Process 160
Part Event 163
Pattern 164
Permit 166
Permit Process 169
Petition 172
Place 174
Primitive 177
Primitive Exception 178
Process 179
Process Event 182
Process Exception 183
Programming Exception 185
Protected 186
Real 187
Reservable Means 188
Resource 189
Same 191
Set 192
Stack 194
String 196

Teleaddress 198
Telename 199
Ticket 200
Ticket Stub 202
Time 203
Trip Exception 205
Uncopied 207
Unmoved 208
Verified 209
Way 210

Appendix: Safety weaknesses 211

Masquerade 211
Leakage or loss of data 211
Denial of service 211
Other weaknesses 212

Index 213

Preface

The personal computer has flourished because it is an open platform. On that platform thousands of independent software developers have built a wealth of standalone applications. Because of their work, anyone—engineer, mathematician, financier, writer, or musician—can find applications that make the personal computer an indispensable tool.

The same cannot be said of networks. Rather than a wealth of distributed, or communicating, applications, one finds for the most part only the familiar remote filing, remote printing, electronic mail, and database applications.

Today's networks—especially public networks—pose an insurmountable barrier to the development of communicating applications. Such applications have a functional need to distribute themselves among the computers of individual users and those that users share, the servers. However, such distribution is impossible for reasons of logistics, portability, and safety.

General Magic has developed a software technology that removes the barrier to developing communicating applications. Conceived for a new breed of consumer electronics product, the personal intelligent communicator, Telescript technology enables a new breed of network, a network that is a highway for *mobile agents* and thereby an open platform for developers.

The economic and social consequences of public networks that are platforms potentially dwarf those of the personal computer. A network equipped for mobile agents makes possible an electronic marketplace in which the *agents* of providers and consumers of goods and services can find and interact with each other. New forms of electronic commerce and community will result.

Introduction

Telescript technology integrates an electronic world of computers and the networks that link them. This world is filled with Telescript places that are occupied by Telescript agents. Each place or agent in the electronic world represents an individual or organization in the physical world, its authority.

Both agents and places are software objects. A place is stationary, but an agent can travel from place to place at will. The agent can do this whether the two places are in the same computer or in different computers. If they are in different computers, a network is involved.

The electronic world of Telescript technology can be called an *electronic marketplace* because within it the agents of providers and consumers of goods and services can find and interact with one another. Thus Telescript technology makes possible a new generation of electronic commerce.

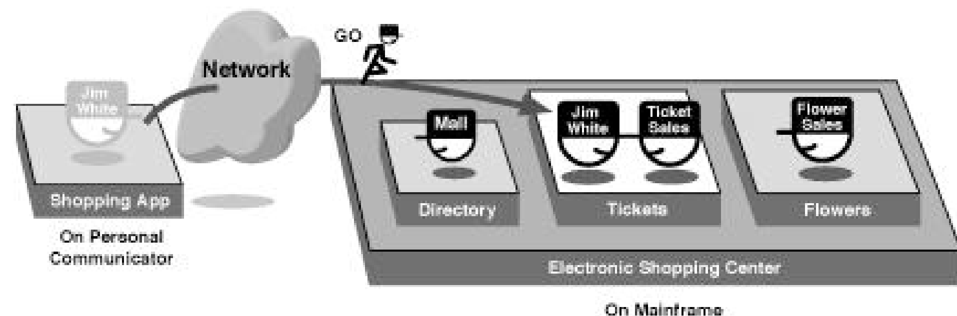
Note: For a fuller exploration of the applications of Telescript technology, see the General Magic white paper, *Telescript Technology: Mobile Agents*

About the technology

This section describes the Telescript model; the Telescript language, which implements the model; the Telescript engine, which implements the language; and the telesphere, which links engines.

Telescript model

The *Telescript model* is the view of computers and computer communication suggested earlier and illustrated here—that is, an electronic world composed of *places*, each occupied by *agents*, which can travel from place to place.



The power represented by an agent's mobility is counterbalanced by *permits*, which a programmer or administrator can use to grant only particular capabilities to particular agents or places on particular occasions.

A place's or agent's *authority* in the physical world is revealed by its *telename*, which it can neither falsify nor withhold from another place or agent. A place, but not an agent, also has a *teleaddress*, which designates its location in the electronic world and reveals the authority of the individual or organization that is operating the computer in which the place exists.

As in the previous illustration, the typical place is permanently occupied by an agent of the place's authority, and is temporarily occupied, or *visited*, by agents of other authorities. For example, a theater ticketing place might be

occupied by a ticketing agent that provides information about theater events, and sells tickets to them as well. Agents of other authorities would visit the ticketing place to use the services the ticketing agent offers.

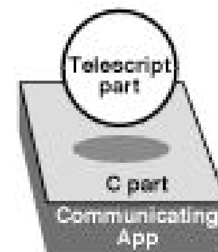
The typical agent travels to obtain a service offered remotely. For example, an agent might go from a place in a user's personal communicator to the ticketing place to obtain theater tickets from the ticketing agent. In general, an agent travels to *meet* and interact with a remote agent. The agents interact programmatically, using object-oriented techniques.

Because agents don't interact at a distance, the model employs remote programming (RP) rather than remote procedure calling (RPC). RP improves upon RPC by enabling computers to interact without communicating; this improves the performance of their interactions by reducing their latency. RP also lets computers customize one another by stationing their own agents—and thus in effect themselves—in each others' domains.

Note: In fact, the need for a place to be permanently occupied by an agent of the place's authority is so common that the model lets this one agent's functionality be incorporated in the place, so that a separate agent isn't required.

Telescript language

The *Telescript language* is an object-oriented remote programming language. It supplements, rather than replaces, systems programming languages like C and C++. As illustrated, only the parts of an application that move from place to place (the agents) or host visiting agents (the places) are written in this language.



The language has the following characteristics:

- *Safety*. The language prevents an agent from directly manipulating its host computer, exceeding its permit, or interacting with other agents without their approval. This helps prevent the spread of viruses.

- *Portability*. The language makes no concessions to the hardware or software constraints or peculiarities of a particular computer. This means that an agent or place can be executed anywhere in a network.

- *Extendability*. The language gives to classes of information object that the programmer defines the stature of classes that are built into the language. This lets the language be extended for particular purposes.

- *Elevation*. The language makes no distinction between volatile and nonvolatile storage. Every information object is persistent. This increases an agent's level of abstraction and decreases its size.

The Telescript language is *communication-centric*. Just as PostScript is designed for describing complex images, and Mathematica for performing complex mathematics, the Telescript language is designed for doing complex networking tasks such as navigation, transportation, and authentication.

The language's `go` operation lets an agent travel. The agent merely presents a *ticket*, which identifies its destination. An agent executes the `go` operation when it needs to get from one place to another. The next instruction in the agent's program is executed at the agent's destination, not at its origin. In a sense, the language reduces networking to one program instruction.

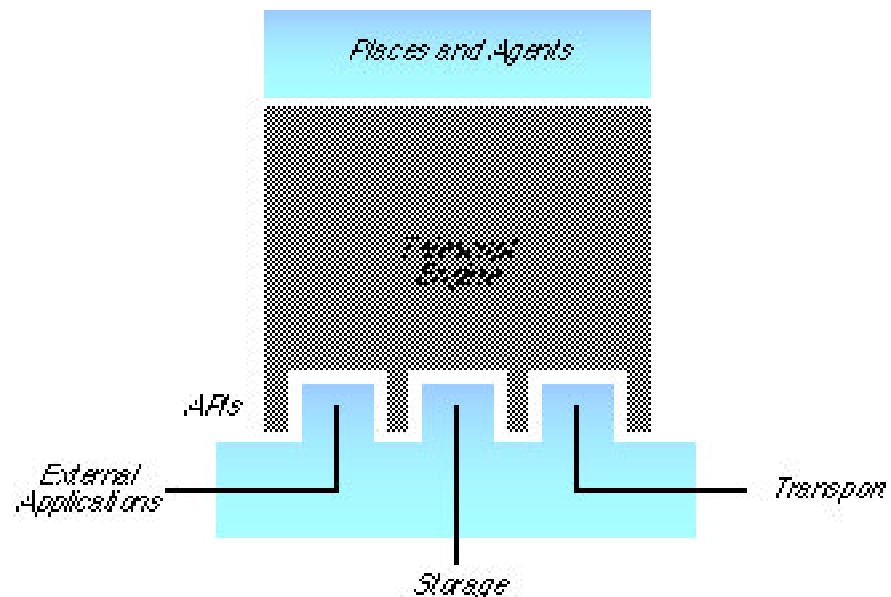
The language's `meet` operation lets two agents meet. One agent presents a *petition*, which identifies the other agent, which must agree to a meeting. An agent executes the `meet` operation when it needs assistance from another agent. By meeting, the two agents receive *references* to one another. The references let the agents interact as peers, using object-oriented techniques.

Notes.

- Because it is for remote programming, the language includes concepts that span the realms of languages, operating systems, and networks. Conventionally separate, in this language these areas are brought together.
- Often, as above, the term *language* refers broadly to both the Telescript programming language and the Telescript predefined classes, which provide much of the technology's functionality (for example, the `go` operation). Sometimes the term refers narrowly to the programming language alone.

Telescript engine

A *Telescript engine* is a computer program that executes the language's object programs. An agent or place is powerless without an engine. An engine can execute two or more—typically many—object programs concurrently.



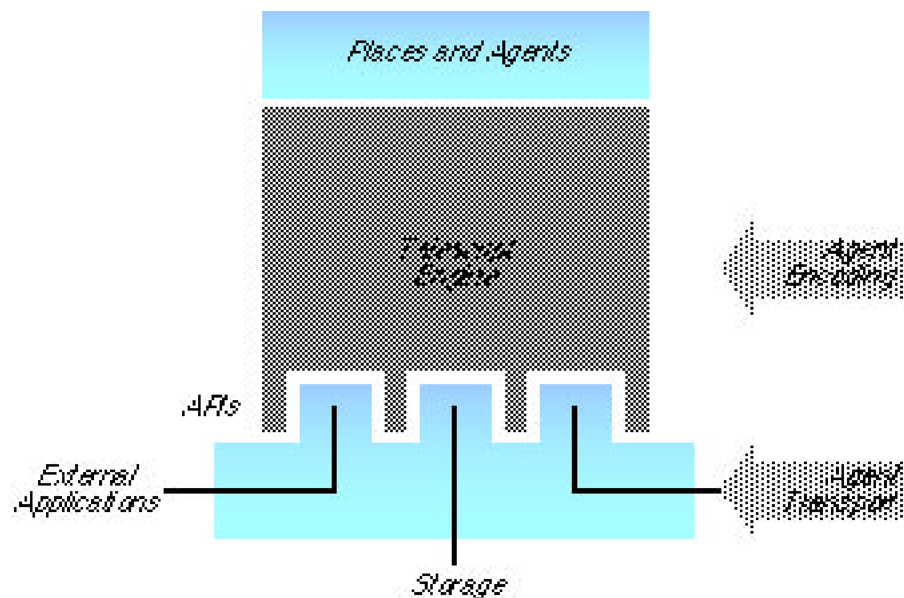
As illustrated, an engine designed for portability draws upon the resources of its computer via application programming interfaces (API s). A storage API provides access to the computer's nonvolatile storage, which the engine uses to preserve places and agents in case of a crash. A transport API provides access to the computer's communication media, which the engine uses to send agents to, and receive agents from, other engines. An external applications API lets the parts of an application written, for example, in C create and interact with the parts written in the Telescript language.

An engine designed to support the places and agents of many authorities implements privileged escapes from the language, which make possible the construction of operational, administrative, and managerial (OAM) tools external to the engine. Such tools are important to the success of large-scale communication systems—for example, those offered as public services.

Telesphere

The *telesphere* is composed of one or more interconnected engines, each of which provides places between which agents can travel, subject to access controls. This requirement makes the telesphere homogeneous with respect to the processing as well as the structure of information.

As illustrated, engines are interconnected so that they can transport agents between them whenever the agents request the `go` operation. An agent is transported as an octet string using suitable communication protocols. The sending engine *encodes* the agent to produce the octet string. Encoding the agent entails encoding the lesser objects of which the agent is constructed. The receiving engine *decodes* the octet string to reconstruct the agent. All engines encode, decode, and execute agents in the same way.



About this manual

This section provides context for the manual.

Audience

This manual is two books under one cover, both written for the Telescript programmer. The first defines the Telescript language itself. The second book defines the Telescript predefined classes. Thus the manual has four parts:

- Part One—Language Concepts
- Part Two—Language
- Part Three—Predefined Class Concepts

- Part Four—Predefined Classes

The manual also has an appendix which lists the known shortcomings of this version of the language and its predefined classes from a safety standpoint.

This reference manual demands much of the reader. A book that makes the technology more widely accessible is being written.

Informal conventions

This manual observes the following informal conventions:

- A class is denoted by its identifier (“Boolean”). If it’s several words (“MeetingPlace”), a space is inserted between words (“Meeting Place”).
- A class member is denoted by its identifier, but with its first character lowercase (“boolean”). If the identifier is several words (“MeetingPlace”), a space is inserted between words (“meeting place”).
- An actual parameter is denoted by its formal parameter followed by the word “parameter” (“the `Item` parameter”).
- An actual argument is denoted by its formal argument followed by the word “argument” (“the `ticket` argument”).
- An attribute is denoted by its identifier followed by the word “attribute” (“the `length` attribute”).
- An operation is denoted by its identifier followed by the word “operation” (“the `copy` operation”).
- An exception is referred to as follows. If *X* is the identifier of Exception or a subclass thereof, “throws *X*” means “throws a member of *X*.”
- As here, the term *identifier* sometimes denotes an identifier’s text, rather than the identifier as a whole. This is a mere economy of expression.

Formal conventions

This manual uses Backus-Naur Form (BNF) to describe strings and octet strings. In such descriptions, the manual observes the following conventions:

- “|” separates syntactic alternatives.
- “`[`” and “`]`” surround a syntactic option.
- ““” and “”” surround a terminal that is a string (rather than an octet string). Whether the terminal is uppercase or lowercase is significant.
- If two or more segments of a string or an octet string could satisfy a nonterminal, the longest segment is taken to satisfy it.
- In prose, for example, “`Match`” is shorthand for “the characters that satisfy the `Match` nonterminal”.

References

This manual incorporates by reference the following documents:

- *The Unicode Standard: Worldwide Character Encoding*, Volume 1, Version 1.0, The Unicode Consortium, Addison-Wesley, 1991.
- *Unicode Technical Report #4, The Unicode Standard*, Version 1.1 (Prepublication Edition), Unicode Inc., 1993.
- *File System Safe UCS Transformation Format (FSS_UTF)*, X/Open Preliminary Specification, Document Number P316, X/Open.
- *IEEE Standard for Binary Floating-Point Arithmetic*, 754-1985 (Reaff 1991), American National Standards Institute (*ANSI*).

Hereafter and in companion documents, the *language manual* is the present document, the *Unicode specification* the first two documents cited above.

Part One— Language Concepts

This part of the manual defines the major concepts—in particular, the object and class abstractions—that underlie the language.

Chapters are devoted to the following topics:

1. object composition, which describes how an object's externally visible interface and externally invisible implementation are defined.

2. object manipulation, which describes how objects are manipulated.

3. object aggregation, which describes how objects with the same interface and implementation are defined collectively rather than individually.

4. object terminology, which introduces some important terminology.

Object composition

An *object* is the unit of both information and information processing. It has an externally visible interface, consisting of operations and attributes, and an externally *invisible* implementation, consisting of methods and properties.

Note: An object can be simple (for example, a boolean) or complex (a dictionary), passive (a string) or active (an agent or a place).

Operations

This section introduces the concept of an operation and describes how an operation's interface and implementation are defined.

The operation concept

An *operation* is a task that one object *performs* at a second's *request*. The object that requests the operation is the *requester*. The object that performs it is the *responder*. The requester and the responder may be the same.

When an operation is requested, the requester provides the responder with zero or more objects, the operation's *actual arguments* (or *arguments*). The first zero or more arguments are *fixed in number*. The remaining zero or more, which the operation may not allow, are *varying in number*. A constraint (see "Constraints") is placed individually upon each argument fixed in number and collectively upon each argument varying in number.

When an operation is performed, it either *succeeds* or *fails*. If the operation succeeds, the responder may *return* to the requester a single object, the operation's *result*. A constraint is placed upon the result. If the operation fails, the responder *throws* to the requester a single object, an *exception*. The requester can *catch* the exception. Unless it does so, the engine *propagates* the exception by effectively causing the requester to throw it.

An operation is private or public. An object performs a *private operation*—to which the engine, rather than the object itself, controls access—only at its own request. An object performs a *public operation* at either its own request or that of another object. (Thus public operations let objects interact.)

An operation that an agent or place performs is sponsored or unsponsored. An agent or place performs a *sponsored operation* under its own authority, which the engine instates at the performance's start and withdraws at the finish. An *unsponsored operation* is performed with no change in authority.

Note: The choice between private and public and the choice between sponsored and unsponsored are orthogonal.

Defining an operation's interface

An operation has an *interface* which dictates how the requester and the responder interact. The interface specifies the following:

- An identifier that distinguishes the operation from the others the object performs and the attributes it gets and sets (see "Defining an attribute").
- How many fixed arguments the requester supplies and the constraints that are placed upon those arguments individually (see "Constraints").

- Whether the requester supplies varying arguments and, if so, the constraint that is placed collectively upon each of those arguments.
- Whether the operation returns a result and, if so, the constraint that is placed upon that result.
- Whether the operation is private or public and, if the responder is an agent or place, whether the operation is sponsored or unsponsored.

Defining an operation's implementation

An operation has an *implementation* which dictates how the responder performs the operation. The implementation takes the form of a *method*. Like the operation it implements, a method is *performed* and *succeeds* or *fails*.

There are three kinds of method. A *predefined method* is part of the engine. A *user-defined method*, written in the language, is part of an object program that the engine interprets. An *out method* is part of system or application software outside the engine and accessible to it by means of its API s.

Notes.

- For example, an object some of whose operations have out methods might provide access to an information service offering news, weather forecasts, stock market results, and other information from outside the engine.
- Because a reference to the same object can be conveyed to any number of agents and places, an object's methods must be prepared in general for the object's operations to be requested any number of times concurrently.

Implementing a method with a block

A *block*, the typical method, is a list of statements (see "Statements") and variable declarations. A *variable declaration* defines the identifiers and type of one or more local variables. A variable declaration may also initialize the variables it introduces by assigning the same object to all of them.

The scope of a local variable is defined in terms of variable declaration segments. A *variable declaration segment* is one or more variable declarations with no adjacent variable declarations and with no intervening statements. A local variable's scope extends from the beginning of the segment in which it's declared to the end of the block. Even if declared in different segments, two local variables declared in the same block shall have different identifiers.

Besides declaring local variables *explicitly* as described above, a block may declare local variables *implicitly*. Any implicit variable declarations effectively precede the first item of the block. A block shall not declare explicitly a local variable whose identifier is that of a local variable it declares implicitly.

A block is *executed* by executing its items from left to right. A statement is executed as discussed elsewhere (see "Statements"). A variable declaration is *executed* by initializing any of its local variables that require initialization. A block's *value* is as follows. If the block's last item is a statement, the block's value is that statement's value. Otherwise the block's value is null.

Notes.

- Statements and variable declarations may be freely intermixed.
- A block's context determines the local variables (if any) it declares implicitly. In particular, the blocks that figure in methods, catchphrases, the `for-to` statement, and the `for-in` statement declare local variables implicitly.

Maintaining local variables

The internal state of one performance of a method is zero or more objects, the method's *local variables*. They're set to nil before the method is performed and discarded after. In between the method can *get* and *set* them.

A constraint is placed upon a local variable just as it is upon an argument or a result. However, the constraint's passage is disregarded and its type is enforced by the compiler but not by this version of the engine.

Attributes

This section introduces the concept of an attribute and describes how an attribute's interface and implementation are defined.

The attribute concept

An *attribute* is an object that one object *gets* or *sets* at a second's *request*. The object that requests the attribute is the *requester*. The object that gets or sets it is the *responder*. The requester and the responder may be the same.

An attribute is the product of two operations, its getter and setter, which the attribute's requester requests and the attribute's responder performs. The *getter* gets the attribute. It has no arguments but has the attribute as its result. The *setter* sets the attribute. It has no result but has the attribute as its sole argument, an argument fixed in number. The same constraint (see "Constraints") is placed upon the getter's result and the setter's argument.

An attribute may be read only. A *read only* attribute—to which the engine, rather than the object itself, controls access—cannot be set. A read only attribute has no setter and so is the product of its getter alone.

Note: Like any operation, the getter or setter is private or public and may be sponsored or unsponsored. The choices made for the getter and the choices made for the setter are orthogonal, although this version of the compiler links them.

Defining an attribute's interface

An attribute has an *interface* which dictates how the requester and the responder interact. The interface specifies the following:

- An identifier which distinguishes the attribute from the others the object gets and sets and the operations it performs.
- Whether the attribute is read only.
- The constraint placed upon the getter's result. (Unless the attribute is read only, the setter's argument is subject to the same constraint.)
- Whether the getter is private or public and, if the responder is an agent or place, whether the getter is sponsored or unsponsored.
- Unless the attribute is read only, whether the setter is private or public and, if the responder is an agent or place, whether the setter is sponsored or unsponsored.

Note: The manual sometimes speaks as if an attribute's getter and setter had different identifiers. These different identifiers are fictions of the exposition.

Defining an attribute's implementation

An attribute has an *implementation* which dictates how the responder gets and sets the attribute. The implementation takes the form of a method for the attribute's getter and, unless the attribute is read only, one for its setter.

An attribute's getter and setter may have predefined methods that maintain the attribute as a property of the responder (see "Maintaining properties"). The property has the attribute's identifier. The setter discards the property and substitutes its argument for it. The getter returns the property as its result, thus giving to its requester the object that the setter received most recently.

A predefined attribute has the above predefined methods for its getter and setter if it isn't read only. The manual notes the few exceptions to this rule.

A user-defined attribute has the above predefined methods for its getter and setter according to the following rule, which requires information in later subsections: a concrete user-defined class has predefined methods for the getter and setter of a user-defined instance attribute if, collectively, the class and its implementation superclasses provide a method for neither operation.

Notes.

- Getting the typical attribute—unless it's passed *byCopy* (see "Defining a constraint's passage")—exposes its property to ongoing examination.
- Getting the typical attribute—unless the attribute is passed *byCopy* or *byProtectedRef* or the responder accesses the property with a protected reference—exposes its property to ongoing modification.
- If the typical attribute is required by its type to be, for example, a member of Event, and an instance of a subclass of Event is given to the setter, the getter later returns that instance of that subclass, not an instance of Event.

Maintaining properties

An object's internal state is zero or more objects, its *properties*. They're set to nil before the constructor is performed and discarded when the object is destroyed. In between the methods the object performs can *get* and *set* them.

Each of an object's implementation member classes (see "Specifying a class's implementation") defines zero or more of the object's properties. The methods that a particular implementation member class defines can get and set the properties defined by that class but not those defined by other classes.

A constraint is placed upon a property just as it is upon an argument or a result. However, the constraint's passage is disregarded and its type is enforced by the compiler but not by this version of the engine.

Constraints

This section introduces the concept of a constraint and describes how a constraint's type and passage are defined.

The constraint concept

A *constraint* dictates an object's type as well as its passage between an operation's requester and responder. An object subject to the constraint is checked (statically by the compiler and dynamically by the engine) to ensure that it satisfies the specified type. Then the object is passed as specified.

Note: Constraints are imposed upon attributes, the arguments and results of operations, and properties; types alone are imposed upon local variables and upon objects in certain other roles in statements and expressions. However, the passage of a property is disregarded and the type of a property or a local variable is enforced by the compiler but not by this version of the engine.

Defining a constraint's type

A *type*, a generalization of a class, is a predicate that certain objects *satisfy*. A type identifies, by means of a class specifier (see "Specifying a class"), a *base class*, whose instances satisfy the type, and two booleans, *isSubclassOK* and *isNilOK*, which may identify other objects that satisfy the type. If the *isSubclassOK* boolean is *true*, instances of interface subclasses of the base class satisfy the type. If the *isNilOK* boolean is *true*, nil satisfies the type.

Types are related in the following ways. A *supertype* of one type is a second type, satisfied by the objects that satisfy the first type, but perhaps satisfied by other objects as well. The first type is a *subtype* of the second. One type is *compatible* with another if the first type's base class is compatible with the second's. A class is *compatible* with itself and with its interface superclasses.

Defining a constraint's passage

A form of *passage* dictates how the engine conveys an object between an operation's requester and responder (including a getter's or setter's). The forms of passage are defined in terms of references (see "Object references").

The following table defines the forms of passage. Throughout the table, *S* is the reference that the object's source gives to the engine. *D*, which the engine derives from *S*, is the reference the engine gives to the object's destination.

Passage	*Definition*
byRef	*D* is *S*.
byProtectedRef	*D* is a protected reference to the referent of *S*.
byUnprotectedRef	*D* is *S*. However, if *S* is a protected reference, the engine throws Reference Protected.
byCopy	The engine copies the referent of *S* and passes *byOwner* the resulting reference to the resulting copy.
byOwner	*D* is *S*. However, unless the referent of *S* is an agent or place, the engine transfers ownership of its closure between the current owner and the responder's owner (see "Transferring ownership").

An object's passage is explicit in a constraint and implicit elsewhere in the language. The passage of an operation's argument or result is explicit, that of an operation's responder or thrown exception implicit. An operation's responder is passed *byRef*. A thrown exception is passed *byOwner*. However, a *lightweight primitive* is always passed *byCopy* no matter what its passage.

Specifying a class

A *class specifier* denotes a class using identifiers. If the class is derived from a class family, an identifier denotes the class family, a class specifier each parameter used in the derivation. Otherwise an identifier denotes the class.

The identifier used in a class specifier denotes a class listed below. Each list item describes zero or more classes to which identifiers are bound. The items appear in order of decreasing precedence. Of all the classes the list includes, the identifier denotes the one of highest precedence with that identifier:

1. If the current class (see “Current objects”) was derived from a class family, the classes that constitute the parameters used in the derivation.

2. If the current class or its class family was defined in a module, the classes in the module (which include the current class or its class family).

3. The predefined classes.

Thus the compiler knows of the predefined classes. In fact, the compiler exhibits knowledge of certain predefined features beyond their signatures. For example, the predefined `copy` operation’s signature is Object →Object but the compiler regards it as (λt.(t →t))(*obj*.`class`), for any object, *obj*.

Note: Further examples are given elsewhere (see “Applying an infix operator”).

Searching for a class

A class specifier identifies a class to the compiler. Sometimes a class must be identified to the engine so that the engine can find it at runtime. The engine must find classes upon which classes to be constructed depend, the classes of which instances are to be constructed, and other classes in other situations.

While classes, like other objects, can be exchanged as the arguments and results of operations, the engine provides a more systematic means of class distribution and uses it for the above purposes. When offered a type and zero or more members of the predefined Package class, the *class search algorithm* looks among the following classes for one that satisfies the type. If one or more satisfy the type, the algorithm produces one. Otherwise it produces nil:

- The predefined classes.

- The values of the packages.

- The classes that can be derived from the class families among the above.

Object manipulation

An object is accessed with a reference. Once accessible in this fashion an object can be manipulated in several ways. An object is owned by an agent or place. An object can be manipulated in some ways only by its owner.

Object references

This section introduces the concept of a reference and describes how a reference can let an object be examined, examined and modified, or neither.

The reference concept

A *reference* is what provides access to an object, its *referent*. There is at least one reference to every object, but there may be any number of them.

Notes.

- The predefined `new` operation returns a reference to a new object. Many other predefined operations return references to new or existing objects.
- The predefined `isSame` operation discloses whether two references provide access to the same object, that is, have the same referent.

Acquiring a protected reference

A reference is either protected or unprotected. A *protected reference* lets its referent be examined. An *unprotected reference* lets its referent be examined and modified. To *examine* an object is to obtain a reference to an object in its closure. To *modify* an object is to modify or replace such an object. An object's *closure* is the object, the object's properties, and their properties, recursively.

An operation whose responder is accessed by a protected reference is limited as follows. If the operation would examine the responder, the engine replaces with a protected reference the reference otherwise obtained. This prevents the operation's requester from modifying the responder later. If the operation would modify the responder, the engine throws Reference Protected.

Notes.

- The predefined `new` operation returns an unprotected reference to a new object unless the object is a member of the predefined Protected class.
- The predefined `isProtected` attribute discloses whether a reference is protected. The predefined `protect` operation exchanges an unprotected reference for a protected one. (Of course, the reverse exchange is impossible.)
- Modifications made to an object (by means of an unprotected reference) are visible by means of all references to that object, protected or unprotected.
- In normal situations, objects ultimately modify *themselves*. In special situations, an object is modified by another object or by the engine itself. Isolation, locking, freezing, and thawing are among these special situations.

Acquiring a voided reference

A *voided reference* no longer provides access to its referent. A reference is voided in circumstances prescribed by the predefined classes, some of which make the voiding appear spontaneous to the object that holds the reference.

A reference is voided, among other reasons, if it would span too great a distance in the place hierarchy established by the predefined classes. An operation's requester and responder can be owned by one agent or place, by two occupants of one place, or by a place and an occupant. In any other situation, the engine voids the reference used to designate the responder.

Whether the reference to an operation's responder is voided before the operation is requested, when the operation is requested, or during the performance of the requested operation, the engine throws Reference Void. However, the predefined `discard` operation doesn't throw this exception.

Protecting objects

A *protected object*, a member of the predefined Protected class, cannot be modified because *all* references to it are protected references.

A protected object is unlocked while it performs the constructor (see "Performing the constructor"). This lets the object initialize itself. The object is locked (see "Locking an object") when the `new` operation succeeds.

Notes.

- The predefined `isSame` operation considers two protected objects to be the same object if they're copy-equal. Therefore the two objects *are* the same.
- While a protected object can't be modified in normal situations, it can be modified as a consequence of isolation, freezing, or thawing.
- Locking a protected object doesn't modify it because it's locked already.
- The protected objects include (but are not limited to) members of the predefined Package, Class, and Primitive classes.

Object manipulation

This section defines basic forms of object manipulation.

Note. The most basic forms of object manipulation are examination and modification which are discussed elsewhere (see "Object references").

Copying an object

To *copy* an object is to create another with certain similarities to the original. Two objects are *copy-equal* if one could have been created by copying the other. Interface members of the predefined Uncopied class cannot be copied.

A copy is created from the original as follows. The objects in the closures of the original and the copy can be paired so that one object in each pair is a copy of the other. Let *O* be any object in the closure of the original, and let *O'* be the object in the closure of the copy that is paired with *O*. Let *R* be the reference that links to *O* any of its properties, and let *R'* be the reference that links to *O'* the corresponding property. The copy is created so that *R* and *R'* access paired objects. If these objects are protected or uncopied objects, *R'* is unprotected if and only if *R* is unprotected. Otherwise *R'* is unprotected.

For purposes of copying and thus determining copy-equality, the attributes that the predefined classes define are implemented as instance properties except as noted in specializations of the predefined `copy` operation.

Notes.

- The predefined `copy` operation copies its responder. The predefined `isEqual` operation discloses whether its responder and its argument are either the same object or copy-equal (without distinguishing between the two cases).
- The original and the copy are instances of the same class.
- Agents and places are uncopied objects.

Destroying an object

To *destroy* an object is to void all references to it; discard the object's references to its properties; and, if the object is an agent or a place, destroy the other objects it owns. An object's *size*, which can vary from one engine to another, is the approximate amount of storage in octets thereby released.

An object is not destroyed upon request. The engine destroys any object that would be excluded from its owner's closure if the owner were isolated. For purposes of this rule, an object's properties include the following:

- While performing an operation, an object includes among its properties any argument of that operation that was passed *byOwner*.
- While designated the current owner, an agent or place includes among its properties any object that is created, by construction or by copying, or that is passed *byOwner* as the result of an operation other than a getter.
- While performing a sponsored operation at the engine's request, an agent or place includes among its properties the responder, stack, and local variables for each method entailed by its activation for that purpose.

Notes.

- In the simplest case, the engine destroys an object to which all references have been either voluntarily discarded or forcibly voided.
- The current object can destroy any object it can put in the required position. For example, if the reference that makes an object a property of the current object is the only reference that includes the property in its owner's closure, replacing the property (for example, with nil) provokes its destruction.

Object ownership

This section introduces the concept of object ownership and lists an agent's or place's privileges while the current owner (see "The `own` statement").

The ownership concept

Every object is *owned* by an agent or place. An agent or place owns itself. Any other object is owned by the agent or place that is designated the current owner when the object is created, whether by construction or by copying.

An agent or place can isolate, lock, freeze, and thaw objects it owns. The current object can exercise these ownership privileges of the current owner.

Notes.

- The `owner` global variable provides an unprotected reference to the agent or place designated the current owner. The predefined `isOwned` attribute discloses whether its responder is owned by the current owner. The predefined `owner` attribute, which is private, discloses to an object its owner.

- The predefined Package class grants the current object the additional privilege of constructing new packages authorized by the current owner.
- Ownership isn't to be confused with authority.

Isolating an object

The current object can isolate objects that the current owner owns. To *isolate* an object, *O*, is first to reduce the closure of *O* by voiding any references the closure includes to objects that the owner of *O* doesn't own, which leaves *O* uniformly owned, and then to void all references outside the resulting closure that access objects—other than *O* itself—inside the closure. An object is *uniformly owned* if all objects in its closure have the same owner.

Notes.

- The predefined `isolate` operation isolates its responder.
- An agent or a place, like any other object, can be isolated. The predefined `partAll` operation, among other things, isolates an agent.

Locking an object

The current object can lock objects that the current owner owns. To *lock* an object is first to isolate it and then to make it subsequently treat unprotected references to itself as though they were protected references.

Notes.

- The predefined `lock` operation locks its responder. Although a locked object can't be unlocked, the predefined `unlockedCopy` operation creates an unlocked copy of its responder even if the responder is locked. Finally, the predefined `isLocked` attribute discloses whether its responder is locked.
- Locking an object doesn't lock its properties.
- Locking an object prevents its subsequent modification.

Freezing an object

The current object can freeze objects that the current owner owns. To *freeze* a locked object is to discard its properties. This is done in the presence of a member of the predefined Package class whose values include an adequate substitute for the object. A class is an adequate substitute for itself. An adequate substitute for any other object is an unfrozen object that matches.

An object is dysfunctional while frozen. The engine throws Object Frozen if the object is asked to perform any but a handful of predefined operations. If the object is a class, the engine throws the same exception if the operation is requested of a member of the class. The operations that a frozen object or a member of a frozen class *can* perform are the `isolate` and `protect` operations and the getters for the `isLocked`, `isOwned`, `isProtected`, and `isFrozen` attributes (the last of which reveals whether the object is frozen).

Notes.

- The predefined `freeze` operation freezes objects. The predefined `isFrozen` attribute discloses whether its responder is frozen.
- A frozen object is a placeholder for the unfrozen object, dramatically less functional but often dramatically smaller in size in return. To reduce its size in transit, an agent can freeze some of the objects it owns just before it departs on a trip and can thaw them just after it arrives at its destination.

Thawing an object

The current object can thaw objects that the current owner owns. To *thaw* a frozen object is to instate as its properties those of another object. This is done in the presence of a member of the predefined Package class whose values include an adequate substitute for the object. A class is an adequate substitute for itself. An adequate substitute for any other object is a value whose key and the name of whose package are copy-equal to the key and the name of the package of the value that proved suitable at freezing.

Notes.

- The predefined `thaw` operation thaws objects.
- In practice, a thawed object is copy-equal to the object that was frozen.

Transferring ownership

The current object can acquire or relinquish ownership of an object and its closure, on behalf of the current owner, by passing the object *byOwner*.

Passing *byOwner* an object, *O*, other than an agent or place transfers ownership of its closure. If *O* is an argument, ownership is transferred from the current owner (see “Current objects”) to the responder’s owner. If *O* is the result of an operation other than a getter, ownership is transferred from the responder’s owner to the current owner. However, in either case, if neither of the two uniformly owns *O*, the engine throws Object Unowned. If *O* is a getter’s result, the engine behaves as though its passage were *byRef*

Object aggregation

As an economy, objects with the same interface and implementation are defined and implemented together as a class. As a further economy, classes whose members have different but systematically similar interfaces and implementations are defined and implemented together as a class family.

Class definitions

This section introduces the concepts of a class and a class family, describes how a class's interface and implementation are defined, and introduces many ancillary concepts, most of which have to do with inheritance.

The class concept

A *class* determines a set of objects, its *instances*, and specifies the interface and implementation of each of them. An object's *interface* or *implementation* consists of the interfaces or implementations of its features. An object's *features* are the operations it performs and the attributes it gets and sets.

A class, an object itself, specifies its own interface and implementation as well as those of its instances. Thus a class specifies class and instance features, class and instance methods, and instance properties. (There are no class properties because classes, once constructed, are immutable.) A *class feature* is requested of the class, an *instance feature* of an instance thereof. A *class method* is performed by the class, an *instance method* by an instance thereof. An *instance property* is maintained by an instance of a class.

A class is either predefined or user-defined. A *predefined class* is a universally available extension to the language. It represents a kind of object available to every Telescript programmer. A *user-defined class* is defined by a particular Telescript programmer. It extends the language for a particular purpose.

Note: Instance features or methods far outnumber class features or methods.

The class family concept

A *class family* is a class-producing template or function. The function is defined using one or more identifiers, the *formal parameters* of the class family. The function is applied with the aid of as many classes, the *actual parameters* (or *parameters*) of the *derived class*. Both defined by lists, the formal and actual parameters agree in order as well as number. To apply the function is to substitute the actual parameters for the formal ones.

Besides a source of classes, a class family is itself a class, the one derived using *default actual parameters* (or *default parameters*). If the class family is predefined and an actual parameter is limited to a certain class or interface subclass thereof, the default parameter is that class. Otherwise it's Object.

Specifying a class's interface

Each of the two interfaces that a class *specifies* is the composite of one native interface and zero or more inherited interfaces. The class *defines* the *native interface* and *inherits* the *inherited interfaces* which other classes specify.

The following terms describe the relationships among classes. A class is an *immediate interface subclass* of the classes that specify the interfaces it inherits. The class is an *interface subclass* of the classes that define the

native interfaces of which the inherited interfaces are composed. A class is an *immediate interface superclass* of its immediate interface subclasses, an *interface superclass* of its interface subclasses.

The following terms describe the relationships between classes and objects. An object is an *interface member* of its class and its interface superclasses, all of which are *interface member classes* of the object.

Notes.

- Object has no immediate interface superclasses.
- An object's interface member classes collectively define all of it operations and attributes, both predefined and user-defined.
- This section is the same as the next except that "interface" appears in one and "implementation" in the other.

Specifying a class's implementation

Each of the two implementations that a class *specifies* is the composite of one native implementation and zero or more inherited implementations. The class *defines* the *native implementation* and *inherits* the *inherited implementations* which other classes specify.

The following terms describe the relationships among classes. A class is an *immediate implementation subclass* of the classes that specify the implementations it inherits. The class is an *implementation subclass* of the classes that define the native implementations of which the inherited implementations are composed. A class is an *immediate implementation superclass* of its immediate implementation subclasses, an *implementation superclass* of its implementation subclasses.

The following terms describe the relationships between classes and objects. An object is an *implementation member* of its class and its implementation superclasses, all of which are *implementation member classes* of the object.

Notes.

- Object has no immediate implementation superclasses.
- An object's implementation member classes collectively define all of it methods and properties, both predefined and user-defined.
- This section is the same as the previous except that "interface" appears in one and "implementation" in the other.

Elaborating upon inheritance

A class can be a *mix-in* rather than a *flavor*. A mix-in cannot have instances. Every immediate subclass of a mix-in is a subclass of a class the mix-in designates as its *anchor*. The anchor or its anchor, recursively, is a flavor.

A class or feature can be *sealed*. A sealed class cannot have user-defined immediate subclasses. A sealed feature cannot be implemented by user-defined subclasses of the class that defines it. A class is sealed by itself. A feature is sealed by a class that defines or inherits an implementation of it.

A class or feature can be *abstract* rather than *concrete*. An abstract class cannot have instances. Hence the class and its implementation superclasses need not collectively implement all features that the class defines or inherits. An abstract feature cannot be implemented by the class that defines it.

Notes.

- Flavors provide single inheritance, mix-ins a limited form of multiple inheritance. Informally, flavors act as nouns, mix-ins as adjectives.
- A flavor is either abstract or concrete. A mix-in is effectively abstract.

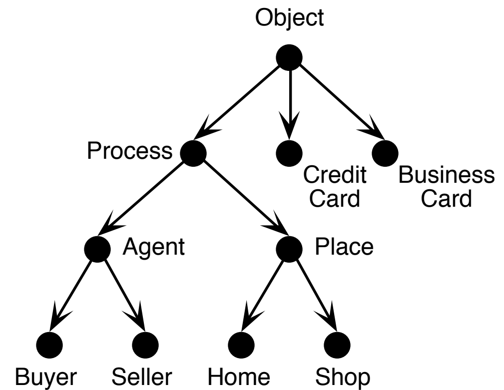
Class relationships

This section describes how classes are related to one another. It introduces the concepts of the class graph, canonical order, and escalation.

Relating one flavor to another

A tree, illustrated, defines the interface inheritance relationships among flavors, a second tree the implementation inheritance relationships.

Either tree is interpreted as follows. The tree's nodes represent flavors, the arc between two nodes the (interface or implementation) inheritance relationship between those two flavors. The tree's root represents Object.



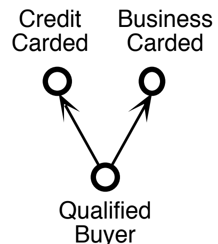
Each destination node reached by an arc that emanates from any source node represents an immediate subclass of the flavor the source node represents. Thus the source node represents an immediate superclass of the flavor the destination node represents.

Each destination node reached by one or more arcs in succession represents a subclass of the flavor the source node represents. Thus the source node represents a superclass of the flavor the destination node represents.

Note. This version of the compiler and this version of the engine require that the interface tree and the implementation tree be one and the same.

Relating one mix-in to another

A tree, illustrated, defines the interface inheritance relationships among a mix-in and its zero or more interface superclasses, a second tree the implementation inheritance relationships.

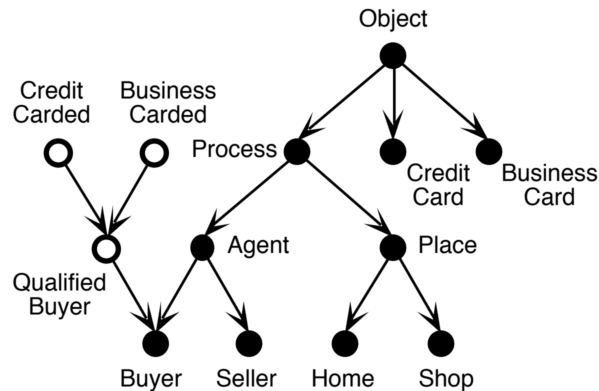


Either tree is interpreted as follows. The tree's nodes represent mix-ins, the arc between two nodes the *inverse* (interface or implementation) inheritance relationship between those two mix-ins. The tree's root represents the mix-in.

Note. This version of the compiler and this version of the engine require that the interface tree and the implementation tree be one and the same.

Relating one class to another

A directed graph, illustrated, defines the interface inheritance relationships among classes—predefined and user-defined classes, flavors and mix-ins. A second graph defines the implementation inheritance relationships.



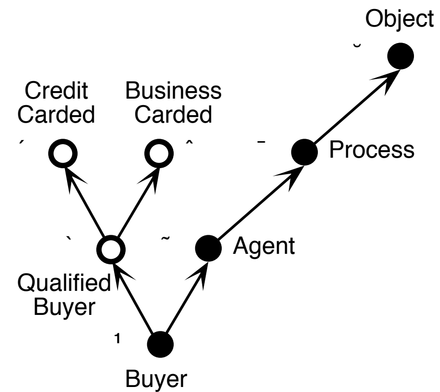
This *class graph* is constructed in two steps. First, each arc in each mix-in tree is reoriented to represent the inheritance relationship, rather than its inverse. Second, the altered mix-in trees are grafted onto the flavor tree.

Notes.

- This version of the compiler and this version of the engine require that the interface graph and the implementation graph be one and the same.
- The language provides the basis for one universal class graph. However, the graph changes over time and is known only in part at any particular place.

Ordering a class and its superclasses

A tree, illustrated, defines the *canonical order* of a class and its interface superclasses, a second tree that of a class and its implementation superclasses.



Either tree is interpreted as follows. The root represents the class, its other nodes the (interface or implementation) superclasses, and the arc between two nodes the inverse inheritance relationship between the two classes.

The canonical order is a depth-first walk of the tree in which a class's immediate superclasses are visited in *their canonical order* (see "Defining a class's immediate superclasses").

Note. This version of the compiler and this version of the engine require that the interface tree and the implementation tree be one and the same.

Searching a class and its superclasses

An object's class and its implementation superclasses can define methods for an operation that the object performs. Thus several methods for the same operation can (and often do) arise. The engine selects for an operation the first method it finds by searching the following four locations in order:

1. If the responder is a class, its class methods.
2. If the responder is a class, the class methods of its implementation superclasses, searched in canonical order.
3. The instance methods of the responder's class.

4. The instance methods of the implementation superclasses of the responder's class, searched in canonical order.

A method for an operation can request either that same operation again or another operation of the same object in a way that constrains the choice of methods. In either case, the operation is said to be *escalated*. In the former case, the search for a method begins immediately after the current class (see "Current objects"). In the latter case, the search begins with a specified class, either the current class or an immediate implementation superclass.

The constructor

This section introduces the concept of the constructor and describes how the constructor differs from other operations.

The constructor concept

An object is *constructed* by asking its class, an instance of Class, to perform the predefined `new` operation. If all actual arguments (if any) are nil, the object is a *basic instance* of its class.The class in turn asks the object under construction to perform the predefined `initialize` operation, often called the *constructor*, whose main purpose is to initialize the object's properties.

Because of its special role in the construction of objects, the constructor differs from other operations in several important respects. These differences help ensure that object construction is orderly and predictable.

Note: Not every object can be constructed using the `new` operation. Some of the objects that cannot be so constructed are literals of the language. Others are created by constructing and then modifying other instances of their classes.

Deciding the constructor's formal arguments

The constructor, defined by Object, is redefined by every other class. A class redefines the constructor's arguments so that using them the class's method for the constructor can initialize the class's properties of the responder.

A class other than Object redefines the constructor either explicitly or implicitly. A flavor's implicitly redefined arguments are the (explicitly or implicitly) redefined arguments of the flavor among the flavor's immediate interface superclasses. A mix-in's implicitly redefined arguments are none.

Deciding the constructor's actual arguments

The constructor is escalated by all methods for it except that defined by Object. This lets all implementation member classes of the object under construction initialize the object's properties that they define. If a method for the constructor fails to escalate it, the engine throws Object Uninitialized.

A class's method for the constructor escalates it with arguments that meet the combined expectations of *all* immediate implementation superclasses—as if the superclasses had declared the constructor's arguments collectively rather than individually. Arguments occur in the collective declaration in the order in which they occur in their individual declarations. Superclasses contribute their arguments to the collective declaration in canonical order.

Performing the constructor

The constructor is performed to the exclusion of other operations. If an object requests an operation of an object that is performing the constructor and

either the requester and responder differ or the method making the request has not escalated the constructor, the engine throws Object Uninitialized.

The constructor's performance affects the selection of methods for other operations. If a method for the constructor, after escalating the constructor, requests another operation of its responder, that operation is implicitly escalated. The current class is the starting point for the method search.

The engine, by throwing Feature Unavailable, prevents the constructor from being requested or escalated other than as described in this section.

Note: A method for the constructor can freely use the features of other objects.

Object terminology

At any point in a program's execution, certain objects play roles sufficiently important to warrant the introduction of terminology for them. The following table lists these objects and the terms the manual uses to refer to them.

Term	*Definition*
current object	The object that's performing the current operation.
current class	The class that defines the current method. However, during the construction of a class, that class.
current operation	The operation that's being performed most immediately.
current method	The user-defined (and not the predefined) method that's being performed most immediately.
current stack	The stack for the performance of the current method. The stack is relevant only to telescript escapes.
current process	The agent or place performing the method for a sponsored operation that most *loosely* encloses the current method.
current sponsor	The agent or place performing the method for a sponsored operation that most *tightly* encloses the current method.
current owner	The owner of the object requesting performance of the predefined `own` operation most tightly enclosing the current method. If there's none, the current process.

Part Two—Language

This part of the manual defines the Telescript language.[1]

Chapters are devoted to the following topics:

1. basic constructs, a brief introduction to the language.
2. definitions, which include those of modules.
3. statements, which are the constituents of methods.
4. expressions, which are the constituents of statements.
5. programs, which typically define modules.

This part of the manual defines the syntax and semantics of Telescript programs. While the syntax rules are given in BNF to the extent possible, many rules are given in prose. Throughout the prose, the word "*shall*" emphasizes a rule whose violation is a compile-time error.

[1] This part of this manual is ©1991-1995 AT&T and General Magic, Inc.

Basic constructs

The Telescript language provides the following constructs among others. A discussion of these constructs provides a brief introduction to the language.

Statements and expressions

The language provides both statements and expressions. A *statement* is *executed* to take actions and possibly to produce a value. An *expression* is *evaluated* to produce a value and possibly to take actions.

A statement's or expression's *value* either is a reference to an object or is *null*, the latter signifying the absence of an object. Where the nature of the reference is irrelevant or is known a priori, the manual uses "value" as if the term denoted the object, rather than a particular reference to the object.

An expression generally combines other, lesser expressions. Evaluating the overall expression entails evaluating some or all of the subexpressions. In general, the order in which subexpressions are evaluated is undefined.

Note: The distinction between statements and expressions is intentionally blurred. Any statement can serve as an expression provided it's enclosed in parentheses. Any expression can serve as a statement directly.

Operation and cascade requests

The language's fundamental combining form, "`x.f(y, z, ...)`", requests an operation, *f*, of an object, *x*, the responder, and supplies *x* with zero or more objects, *y*, *z*, etc., the arguments of *f*. Most other combining forms provide more conventional or convenient notation for specific operations or situations.

Among the other combining forms is one that requests a cascade. A *cascade* is a sequence of two or more operations performed by the same object which is produced (by expression evaluation) only once. The cascade's *responder* is the object that performs all of the operations. The cascade's *result* is that of the operation performed last. If that operation has no result, the cascade has no result. Any results of preceding operations are discarded.

Objects and their identifiers

Many objects are accessed, and in some cases assigned, with *identifiers*. The following table lists categories of identified object and objects in each category. For each category, the identifiers of the listed objects shall differ.

Category	*Objects in category with different identifiers*
Global variable	All global variables.
Local variable	All local variables defined by a block.
Property	All properties defined by a class.
Class	All classes defined by a module, or all predefined classes.
Parameter	All formal parameters defined by a class family.

Class feature	All class features defined by a class or an interface superclass. This version of the compiler doesn't allow an identifier to denote both a class and an instance feature.
Instance feature	All instance features defined by a class or an interface superclass. This version of the compiler doesn't allow an identifier to denote both a class and an instance feature.
Argument	All formal arguments defined by an operation.

The first character of an identifier's *text*, which is a string, shall be an alphabetic character. Each of the text's remaining characters shall be anything but a control, private use, punctuation, space, or special character.

Notes.

- The predefined classes use identifiers for additional purposes.
- The compiler can assign an identifier whose text's first character is "_".

Global variables

Among the objects accessed with identifiers are global variables. A *global variable* is an object that provides context for the evaluation of an expression (see "Expressions"). Global variables are assigned by the engine and accessed (see "Accessing an object") by user-defined methods.

The following table defines the global variables. The first column gives their identifiers, the second gives their definitions, and the third states whether the engine provides protected or unprotected references to them.

Identifier	*Global variable*	*Reference protected?*
`client`	The current operation's requester or, if the requester's the engine, nil.	If that used to request the enclosing operation was.
`self`	The current object.	If that used to request the current operation was.
`here`	The place the current process occupies or, if the current process is the engine place, nil.	No.
`place`	The place the current process occupies or, if the current process is a place, that place.	No.
`process`	The current process.	No.
`sponsor`	The current sponsor.	No.
`owner`	The current owner.	No.

Definitions

The typical Telescript program defines a *module*, a member of the predefined Package class. A module's values are classes, its keys their identifiers.

Defining a module entails defining the module's classes, the interfaces upon which the definitions of the classes depend, the attributes and operations that the classes define, and the constraints placed upon the attributes and upon the arguments and the results of the operations.

Note. A package is both a dictionary and a protected object. The constructor locks a package's keys and values before it adds them to the package.

Module definitions

A module is defined as follows:

```
       Module ::=  ID ":" "module" "=" "(" [ModuleItems] ")"

  ModuleItems ::=  ModuleItem
                 | ModuleItems ";" ModuleItem
                 | ModuleItems ";"

   ModuleItem ::=  Interface | Class | Escape
```

The module is formed as follows. An `Interface` adds nothing to the module. A `Class` adds its value. An `Escape` (see "Escape expressions") adds to the module as would zero or more `Interface`'s and `Class`'s.

All `ModuleItems` shall introduce different identifiers, except that a definition of the interface of a class can precede a backward compatible definition of the class itself. One `ModuleItem` can use the identifier another `ModuleItem` introduces without regard for the lexical order of the two `ModuleItems`.

This is an expression (see "Expressions") whose value is the module.

Notes.

- The classes in the typical module are functionally related (for example, they implement a single communicating application).
- In this version of the language, a class in one module can use the identifier of a class in another only if the first module redundantly defines the second class's interface. The redundant interface definition is unauthenticated.
- One may view the "`;`" as either a separator or a terminator. It is mandatory between `ModuleItems` but is optional after the last `ModuleItem`.

Interface definitions

An interface is defined as follows:

```
    Interface ::=  ID ":" ["sealed"] ["abstract"|"mixin"]
                   "interface" [FormalParameters]
                   [Superclasses] "=" "(" [Features] ")"
```

`Interface` defines a class as would `Class` (see "Defining a class"), but may not define all of the properties and methods that `Class` would define.

This is an expression (see "Expressions") whose value is null.

Note. An `Interface` is easily included in any number of `Module`'s with the aid of the C preprocessor's `#include` directive.

Class definitions

Defining a class entails defining the class itself; if the class is a class family, its formal parameters; the class's immediate superclasses; and the class's features. Defining a class also entails identifying any sealed features.

Defining a class

A class is defined as follows:

```
Class ::=  ID ":" ["sealed"] ["abstract"|"mixin"]
           "class" [FormalParameters]
           [Superclasses] "=" "(" [Features] ")"
```

In the discussion that follows, let *C* be the defined class. If *C* is a class family, let *D* be any class derived from *C*. Otherwise let *D* be *C* itself.

C and *D* are as follows. `ID` denotes *C* for purposes of a `ClassID`. If "`sealed`" occurs, *D* is sealed. If "`abstract`" occurs, *D*, a flavor, is abstract not concrete. If "`mixin`" occurs, *D* is a mix-in. If `FormalParameters` occurs, it's the formal parameters of *C*, a class family. Otherwise *C* isn't a class family. If `Superclasses` occurs, it denotes the immediate superclasses of *D*. Otherwise they're Object alone. If `Features` occurs, it's the features, but also the methods and properties, that *D* defines. Otherwise *D* defines none.

To find each class upon which *C* depends (for example, one of its immediate implementation superclasses), the engine follows the class search algorithm using packages that the requester's owner offers privately (see the class manual). If the algorithm fails, the engine throws Class Unavailable.

This is an expression (see "Expressions"). If `Class` occurs as a `ModuleItem`, its value is *C*. Otherwise its value is a module that includes *C* alone.

Defining a class's formal parameters

A class's formal parameters are defined as follows:

```
FormalParameters ::=  "[" FormalGroups "]"

    FormalGroups ::=  FormalGroup | FormalGroups ";" FormalGroup
     FormalGroup ::=  Identifiers ":" "Class" ["<:" ClassID]

     Identifiers ::=  ID | Identifiers "," ID
```

The formal parameters are the `Identifiers` in the `FormalGroups`. If `ClassID` occurs in a particular `FormalGroup`, each actual parameter whose formal parameter is among the `Identifiers` in that `FormalGroup` shall be the class that `ClassID` denotes or an interface subclass thereof.

Notes.

- The compiler but not the engine knows of any such parameter constraints.

- A derived class's functionality can depend upon the actual parameters used in its derivation by the mechanism described in "Defining a class specifier".

Defining a class's immediate superclasses

A class's immediate superclasses are defined as follows. If a class family is being defined, the class is any derived from it:

```
Superclasses ::= "(" ClassIDs ")"
```

`ClassIDs` denotes the immediate interface and implementation superclasses alike in reverse canonical order. However, if the class is a mix-in, the first `ClassID` denotes its anchor. If the class is a flavor, the first `ClassID` shall denote a flavor using an `ID` other than a formal parameter.

Note. In this version of the language, a class's immediate interface and implementation superclasses must be the same classes.

Defining a class's features

A class's features, but also its properties and methods, are defined as follows. If a class family is being defined, the class is any derived from it:

```
Features ::= Feature
           | Features Feature
           | Features ";"

 Feature ::= RequesterDecl
           | ResponderDecl
           | AttributeDecl
           | OperationDecl
           | SealingDecl
```

The compiler processes `Features` from left to right to produce the following effects. `RequesterDecl`'s and `ResponderDecl`'s qualify later `AttributeDecl`'s and `OperationDecl`'s. `AttributeDecl`'s define native attributes, implement native and inherited attributes, and define properties. `OperationDecl`'s define native operations and implement native and inherited operations. `SealingDecl`'s seal inherited features.

Defining features' requesters

Features' requesters are defined as follows:

```
RequesterDecl ::= "private" | "public"
```

`RequesterDecl` leaves *in force* whichever of the two keywords occurs, the keyword "`private`" being in force before the first `RequesterDecl`.

Notes.

- The keywords find use elsewhere (see "Defining an attribute" and "Defining an operation").
- Private features are protected in the C++ sense.

Defining features' responders

Features' responders are defined as follows:

```
ResponderDecl ::= "instance" | "class" | "property"
```

`ResponderDecl` leaves *in force* whichever of the three keywords occurs, the keyword "`instance`" being in force before the first `ResponderDecl`.

Note: The keywords find use elsewhere (the first two in "Defining an attribute" and "Defining an operation", the third in "Defining attributes or properties").

Defining attributes or properties

Attributes or properties are defined as follows:

```
AttributeDecl ::= Identifiers ":" Attribute ";"
```

`AttributeDecl` either declares or redeclares each attribute that `Identifiers` denotes and that `Attribute` defines. A class declares a new, native attribute to define it and perhaps to define a method for its getter, a method for its setter, or both. A class redeclares an existing, inherited attribute to define a method for its getter, a method for its setter, or both.

Any redeclaration of an attribute must be compatible with its declaration (and with any previous redeclaration). The redeclared type of the setter's argument shall be the declared type or a supertype thereof. The redeclared type of the getter's result shall be the declared type or a subtype thereof.

`AttributeDecl` may find the keyword "`property`" in force. If it does, it defines properties, rather than attributes, and all of `Attribute` but the `Type` in its `Constraint` is ignored. A property's type is enforced by the compiler but not by this version of the engine. A property's passage is *byRef*

Note: Properties are private in the C++ sense.

Defining operations

Operations are defined as follows:

```
OperationDecl ::= Identifiers ":" Operation ";"
```

`OperationDecl` either declares or redeclares each operation that `Identifiers` denotes and that `Operation` defines. A class declares a new, native operation to define it and perhaps to define a method for it. A class redeclares an existing, inherited operation to define a method for it.

Any redeclaration of an operation must be compatible with its declaration (and with any previous redeclaration). The redeclared type of any argument shall be the declared type or a supertype thereof. The redeclared type of any result shall be the declared type or a subtype thereof.

`OperationDecl` shall not find the keyword "`property`" in force.

Sealing features

Features are sealed as follows:

```
SealingDecl ::= Identifiers ":" "sealed" ";"
```

`SealingDecl` seals each feature that `Identifiers` denotes. Each feature shall be inherited by, rather than native to, the defined class.

Note: A native feature can be sealed at its declaration. A class seals a feature to prevent the class's interface subclasses from defining methods for it.

Attribute definitions

Defining an attribute entails defining the attribute itself, the attribute's signature, and possibly the attribute's getter and setter.

Defining an attribute

An attribute is defined as follows:

```
Attribute ::=  [“abstract”|“sealed”] [“sponsored”]
               AtSignature [“with” “(” GetterSetters “)”]
```

The interfaces of the attribute's getter and setter are as follows. If “`abstract`” occurs, both operations are abstract not concrete. If “`sealed`” occurs, both are sealed. If “`sponsored`” occurs, both are sponsored. If “`private`” not “`public`” is in force, both are private not public. If “`class`” not “`instance`” is in force, both are performed by the defined class, not an interface member thereof. The attribute's signature is `AtSignature`.

The implementations of the attribute's getter and setter are as follows. If `GetterSetters` occurs, it defines user-defined methods for the getter, the setter, or both. Otherwise none is provided (here).

Notes.

- The typical attribute has predefined, not user-defined, methods.
- In this version of the language, the getter and setter must be categorized together, rather than separately, as either abstract or concrete, sealed or unsealed, sponsored or unsponsored, and private or public.
- The compiler but not the engine knows whether an attribute is abstract.

Defining an attribute's signature

An attribute's *signature* is defined as follows:

```
AtSignature ::=  [“readonly”]
                 Constraint [“throws” ClassIDs]
```

The signatures of the attribute's getter and setter are as follows. `Constraint` constrains both the setter's argument and the getter's result. If “`readonly`” occurs, the attribute has no setter. If `ClassIDs` occurs, it denotes the classes whose interface members the getter, the setter, or both purportedly throw. Otherwise they purportedly throw no exceptions.

Each class that `ClassIDs` denotes shall be Exception or an interface subclass thereof. Furthermore, none of the classes shall be derived from a class family.

Note. In this version of the language, the compiler enforces the second requirement but not the first. The compiler doesn't even insist that the class identifiers be defined.

Note. The engine doesn't have access to the exception class identifiers.

Defining an attribute's getter or setter

An attribute's getter or setter is defined as follows:

```
GetterSetters ::= GetterSetter
                | GetterSetters ";" GetterSetter
                | GetterSetters ";"

GetterSetter ::= ID ":" Operation
```

`GetterSetter` shall occur at most twice. If there are two occurrences, in one, `ID` shall be "`get`" and `Operation` shall define the getter and a method for it. The type of the getter's result shall be compatible with the attribute's type. In the other, `ID` shall be "`set`" and `Operation` shall define the setter and a method for it. The attribute's type shall be compatible with the type of the setter's argument. If there is one occurrence, it shall be either of these two.

If "`abstract`", "`sealed`", or "`sponsored`" occurs in `Operation`, it is ignored.

Note One may view the "`;`" as either a separator or a terminator. It is mandatory between `GetterSetters` but is optional after the last `GetterSetter`.

Operation definitions

Defining an operation entails defining the operation itself; the operation's signature; the operation's expected arguments, named and unnamed; and perhaps the operation's method, which usually takes the form of a block.

Defining an operation

An operation is defined as follows:

```
Operation ::= ["abstract"|"sealed"] ["sponsored"]
              OpSignature ["=" Method]
```

The operation's interface is as follows. If "`abstract`" occurs, the operation is abstract not concrete. If "`sealed`" occurs, it's sealed. If "`sponsored`" occurs, it's sponsored. If "`private`" not "`public`" is in force, it's private not public. If "`class`" not "`instance`" is in force, it's performed by the defined class, not an interface member thereof. The operation's signature is `OpSignature`.

The operation's implementation is as follows. If `Method` occurs, it's a user-defined method for the operation. Otherwise none is provided (here).

Note The compiler but not the engine knows whether an operation is abstract.

Defining an operation's signature

An operation's *signature* is defined as follows:

```
OpSignature ::= "op" "(" [ExpectedArguments] ")"
                [Constraint] ["throws" ClassIDs]
```

The operation's signature is as follows. If `ExpectedArguments` occurs, it constrains and may identify the operation's arguments. Otherwise the operation accepts no arguments. If `Constraint` occurs, it constrains the result. Otherwise the operation returns no result. If `ClassIDs` occurs, it denotes the classes whose interface members the operation purportedly throws. Otherwise it purportedly throws no exceptions.

Each class that `ClassIDs` denotes shall be Exception or an interface subclass thereof. Furthermore, none of the classes shall be derived from a class family.

Note: In this version of the language, the compiler enforces the second requirement but not the first. The compiler doesn't even insist that the class identifiers be defined.

Note: The engine doesn't have access to the exception class identifiers.

Defining an operation's expected arguments

An operation's expected arguments are defined as follows:

```
ExpectedArguments ::=  NamedArguments
                     | UnnamedArguments
                     | FixedGroups [";" UnnamedArguments]
```

The operation's expected arguments are as follows. If `NamedArguments` occurs, it assigns formal arguments to all expected actual arguments. If `UnnamedArguments` occurs, it assigns formal arguments to none. If `FixedGroups` occurs (see "Defining an operation's named arguments"), it assigns formal arguments to all expected actual arguments it encompasses.

Note: Thus the arguments varying in number and the last zero or more arguments fixed in number may (but needn't) be left without formal arguments.

Defining an operation's named arguments

An operation's named arguments are defined as follows:

```
NamedArguments ::=  FixedGroups
                  | VaryingGroup
                  | FixedGroups ";" VaryingGroup

   FixedGroups ::=  FixedGroup | FixedGroups ";" FixedGroup
    FixedGroup ::=  Identifiers ":" Constraint

  VaryingGroup ::=  ID ":" [Constraint] "..."
```

If `FixedGroups` occurs, the operation has arguments fixed in number. The formal arguments are the `Identifiers` in the `FixedGroups`. Each actual argument is subject to the `Constraint` that a `FixedGroup` associates with the `Identifiers` that includes the corresponding formal argument.

If `VaryingGroup` occurs, the operation has arguments varying in number. The formal argument is `ID`. If `Constraint` occurs, it's the constraint upon each actual argument. Otherwise the constraint is defined by "`ref Object`".

A *formal argument* is an identifier that denotes either one actual argument fixed in number or all actual arguments varying in number. Associated with each formal argument is a constraint upon the actual argument it denotes.

The formal arguments of an operation, as defined or redefined by a certain class, shall differ from the identifiers of properties native to that class. The constructor is exempted from this rule (see "Defining an operation's method").

Note: An operation's formal arguments don't affect the interface of the class that defines the operation. However, they do affect the implementation of any class that defines a method for the operation (see "Defining an operation's method").

Defining an operation's unnamed arguments

An operation's unnamed arguments are defined as follows:

```
UnnamedArguments ::= FixedItems
                   | VaryingItem
                   | FixedItems ";" VaryingItem

      FixedItems ::= FixedItem | FixedItems ";" FixedItem
       FixedItem ::= Constraint

     VaryingItem ::= [Constraint] "..."
```

If `FixedItems` occurs, the operation has arguments fixed in number. The number and lexical order of the `Constraint`'s are those of the actual arguments, each of which is subject to the `Constraint` thus paired with it.

If `VaryingItem` occurs, the operation has arguments varying in number. If `Constraint` occurs, it's the constraint upon each actual argument. Otherwise the constraint is defined by "`ref Object`".

Note The use of `UnnamedArguments` is denigrated. Some or all arguments must be unnamed if either the operation's method is an `Escape` not a `Block` (see "Defining an operation's method") or the method escalates the operation with its arguments already on the current stack (see "Escalating the current operation").

Defining an operation's method

An operation's method is defined as follows:

```
Method ::= Block | Escape
```

`Method` is normally `Block` but may be `Escape`, in which case the arguments and result are handled as described elsewhere (see "Escape expressions").

The operation's named arguments are assigned to implicitly declared local variables before `Block` is executed. If there are named arguments fixed in number, `Block` declares one variable for each such argument. The variable's identifier is the formal argument. The variable's type is the type associated with it. If there are named arguments varying in number, `Block` declares one variable for all such arguments collectively. The variable's identifier is the formal argument. The variable's type is that satisfied only by interface members of `List[`*T*`, Equal]`, *T* being the required type of each such argument.

The constructor's named arguments are treated a bit differently. If a formal argument is the identifier of a property of the class that defines the method—even the property that the predefined method for an attribute's getter or setter employs—the argument fixed in number or the list of arguments varying in number is assigned to that property; no local variable is implicitly declared. The property's type shall allow the assignment.

The operation's unnamed arguments are left on the current stack.

The operation's result is as follows. If `Block` executes the `return` statement, that statement determines the result. Otherwise if `Block`'s value isn't null, the result is that value. Otherwise there's no result.

Defining a block

A block is defined as follows:

```
Block ::= "{" [BlockItems] "}"
```

```
BlockItems ::= BlockItem
             | BlockItems “;” BlockItem
             | BlockItems “;”

 BlockItem ::= VariableDecl | Statement

VariableDecl ::= Identifiers “:” Type [“=” Expression]
               | Identifiers “:” “=” Expression
```

The block is as follows. If `BlockItems` occurs, it’s the items of the block. Otherwise there are none. Each `VariableDecl` explicitly declares the local variables that `Identifiers` denotes. If `Type` occurs, the type of each local variable is `Type`. Otherwise the type of each is that of `Expression`’s value. If `Expression` occurs, it’s evaluated once for purposes of initialization.

Notes.

- The relationship between blocks and statements is circular. A block can be constructed of statements. A statement can be constructed of blocks.
- One may view the “`;`” in `BlockItems` as either a separator or a terminator. It is mandatory between `BlockItems` but is optional after the last `BlockItem`.

Constraint definitions

Defining a constraint entails defining the constraint itself, the constraint’s type and passage, and the type’s class specifier.

Defining a constraint

A constraint is defined as follows:

```
Constraint ::= [Passage] Type
```

The constraint is as follows. The type is `Type`. If `Passage` occurs, the passage is `Passage`. Otherwise the passage is *byRef*

In the definition of a predefined but not a user-defined class, a comment that occurs after `Type` formally augments `Type` in this sense: the engine throws Argument Invalid or Result Invalid unless the augmented type is satisfied.

Defining a constraint’s type

A type is defined as follows:

```
Type ::= ClassID [“!”] [“|” “Nil”]
```

The type is as follows. `ClassID` denotes the base class. If “`!`”occurs, the *isSubclassOK* boolean is *false*. If “`|`” occurs, the *isNilOK* boolean is *true*.

Defining a constraint’s passage

A passage is defined as follows:

```
Passage ::= “ref”
          | “protected”
          | “unprotected”
          | “copied”
          | “owned”
```

The passage is *byRef* if "`ref`" occurs, *byProtectedRef* if "`protected`" occurs, *byUnprotectedRef* if "`unprotected`" occurs, *byCopy* if "`copied`" occurs, and *byOwner* if "`owned`" occurs.

Defining a class specifier

A class specifier is defined as follows:

```
 ClassID ::=  ID | ID "[" ClassIDs "]"

ClassIDs ::=  ClassID | ClassIDs "," ClassID
```

The denoted class is as follows. If `ID` occurs alone, the class is the one that it denotes. Otherwise the class is the one derived from the class family that it denotes using as parameters the classes that `ClassIDs` denotes.

`ID` shall denote a class in one of the categories listed elsewhere (see "Specifying a class"). Unless `ID` occurs alone, it shall denote a class family.

Statements

Statements fall conveniently into the following categories. Basic statements evaluate expressions, execute blocks both unconditionally and conditionally, choose among alternative blocks, and provide the results of operations. Iterative statements enable and control indefinite, conditional, and bounded iteration with or without a control variable. Exception statements throw and catch exceptions. Process statements execute blocks under specified conditions of object ownership, object permission, and object synchronization:

```
Statement ::=  BasicStmt
             | IterativeStmt
             | ExceptionStmt
             | ProcessStmt
```

Basic statements

A basic statement is one of the following:

```
BasicStmt ::=  ExprStmt
             | DoStmt
             | IfStmt
             | IfElseStmt
             | ReturnStmt
```

Using an expression as a statement

An expression is used as a statement as follows:

```
ExprStmt ::=  Expression
```

Executing this statement evaluates `Expression`.

The statement's value is `Expression`'s value.

The `do` statement

The `do` statement is written as follows:

```
DoStmt ::=  "do" Block
```

Executing this statement, whose value is null, executes `Block`.

Note: This statement typically introduces local variables with restricted scope.

The `if` statement

The `if` statement is written as follows:

$\texttt{IfStmt}_1$ ::= "if" Expression Block ["else" $\texttt{IfStmt}_2$]

Executing this statement, whose value is null, evaluates `Expression`. The value of `Expression` shall be a boolean. If the value is *true*, `Block` is executed. Otherwise $\texttt{IfStmt}_2$, if it occurs, is executed.

The `if-else` statement

The `if-else` statement is written as follows:

```
IfElseStmt1 ::= "if" Expression Block1 ElseClause

 ElseClause ::= "else" Block2 | "else" IfElseStmt2
```

Executing this statement evaluates `Expression`. The value of `Expression` shall be a boolean. If the value is *true*, `Block`$_1$ is executed. Otherwise `Block`$_2$ or `IfElseStmt`$_2$ is executed.

The statement's value is the executed `Block`'s value. However, if the last statement in any `Block` whose value is null is neither `return`, `throw`, nor a `loop` statement enclosing no `break` statement, the statement's value is null.

The type of the statement's value, if not null, is as follows. Let *S* be the least common interface superclass of the statement's non-null `Block` values. If none of these values is nil, the type is that satisfied only by interface members of *S*. If each of them is nil, the type is that satisfied only by nil. Otherwise the type is that satisfied only by interface members of *S* or by nil.

The `return` statement

The `return` statement is written as follows:

```
ReturnStmt ::= "return" [Expression]
```

Executing this statement, whose value is null, stops the performance of the current operation in such a way that it succeeds. If `Expression` occurs the operation's result is its value. Otherwise there's none. If the operation requires a result, `Expression` shall occur and its value shall not be null.

Iterative statements

An iterative statement is one of the following:

```
IterativeStmt ::= LoopStmt
                | WhileStmt
                | RepeatStmt
                | ForToStmt
                | ForInStmt
                | ContinueStmt
                | BreakStmt
```

An iterative statement either executes a block, the *controlled block*, under specified conditions or stops execution of the controlled block of the most tightly enclosing iterative statement and perhaps that statement itself.

The `loop` statement

The `loop` statement is written as follows:

```
LoopStmt ::= "loop" Block
```

Executing this statement, whose value is null, executes `Block`—the controlled block—repeatedly.

Note: Only a `break`, `return`, or `throw` statement can halt the repetition.

The `while` statement

The `while` statement is written as follows:

```
WhileStmt ::= "while" Expression Block
```

Executing this statement, whose value is null, executes `Block`—the controlled block—repeatedly. `Expression` is evaluated before each execution of `Block`. The value of `Expression` shall be a boolean. If the value is *true*, `Block` is executed. Otherwise the `break` statement is executed.

The `repeat` statement

The `repeat` statement is written as follows:

```
RepeatStmt ::= "repeat" Expression Block
```

Executing this statement, whose value is null, executes `Block`—the controlled block—repeatedly. `Expression` is evaluated once. The value of `Expression` shall be an integer. If the value, *n*, is greater than zero, `Block` is executed *n* times. Otherwise `Block` isn't executed at all.

The `for-to` statement

The `for-to` statement is written as follows:

```
ForToStmt ::= "for" ID [":" Type] "to" Expression Block
```

Executing this statement, whose value is null, executes `Block`—the controlled block—repeatedly. `Expression` is evaluated once. The value of `Expression` shall be an integer. If the value, *n*, is greater than zero, `Block` is executed *n* times. Otherwise `Block` isn't executed at all.

A counter is set to one before the first execution of `Block`, increased by one before every other execution, and assigned to the local variable or property `ID` denotes (see "Accessing an object") before every execution. If `Type` occurs, the `Block` implicitly declares, for this purpose, a variable whose type is `Type`. An integer shall satisfy the type of the specified local variable or property.

Note The new variable and a similarly introduced variable in C++ differ in scope.

The `for-in` statement

The `for-in` statement is written as follows:

```
ForInStmt ::= "for" ID [":" Type] "in" Expression Block
```

Executing this statement, whose value is null, executes `Block`—the controlled block—repeatedly. `Expression` is evaluated once. The value of `Expression` shall be a collection. The collection's predefined `iterator` attribute is gotten. `Block` is executed once for each item of the iterator.

The next item is assigned to the local variable or property `ID` denotes (see "Accessing an object") before every execution of `Block`. If `Type` occurs, the `Block` implicitly declares, for this purpose, a variable whose type is `Type`. An item shall satisfy the type of the specified local variable or property.

Note The new variable and a similarly introduced variable in C++ differ in scope.

The `continue` statement

The `continue` statement is written as follows:

```
ContinueStmt ::= "continue"
```

Executing this statement, whose value is null, stops the execution of the controlled block of the most tightly enclosing iterative statement, but not execution of that statement itself. Any items remaining in the block are skipped.

The `break` statement

The `break` statement is written as follows:

```
BreakStmt ::= "break"
```

Executing this statement, whose value is null, stops the execution of the controlled block of the most tightly enclosing iterative statement, but also execution of that statement itself. Any items remaining in the block are skipped.

Exception statements

An exception statement is one of the following:

```
ExceptionStmt ::= TryStmt
                | ThrowStmt
```

Declaring catchphrases

Catchphrases are written as follows:

```
CatchPhrases ::= CatchPhrase [CatchPhrases]
 CatchPhrase ::= "catch" [ID ":"] Type Block
               | "catch" "(" ID ":" Type ")" Block
```

A *catchphrase*, which is essentially a block, *handles* a thrown exception. The engine *executes* the catchphrase by executing the block. The catchphrase's *value* is the block's. If the block throws an exception, the catchphrase can't handle it (but a catchphrase not at the same level, but at a higher one, can).

A `Catchphrase` can handle any exception that satisfies `Type`. The base class of `Type` shall be Exception or an interface subclass thereof. Its *isSubclassOK* boolean shall be *true*. Its *isNilOK* boolean shall be *false*.

The `Catchphrase` accesses as follows the exception it handles. If `ID` occurs, the exception is assigned to the local variable `ID` denotes (see "Accessing an object") before `Block` is executed. A variable whose type is `Type` is implicitly declared for this purpose. If `ID` doesn't occur, the exception is discarded.

Notes.

- The `try` and `restrict` statements are defined in terms of catchphrases.
- The two forms of `CatchPhrase` serve the same purpose, but with only the first can one avoid implicitly declaring a local variable to receive the exception.

The `try` statement

The `try` statement is written as follows:

```
TryStmt ::= "try" Block CatchPhrases
```

Executing this statement executes `Block` and, if `Block` throws an exception, does the following. If one or more `Catchphrases` can handle the exception, the exception is caught and the lexically first such `Catchphrase` is executed.

The statement's value is the executed `Catchphrase`'s value. However, if no `Catchphrase` is executed, the statement's value is null.

Note: Catching an exception passes it to the current method *byOwner*.

The `throw` statement

The `throw` statement is written as follows:

```
ThrowStmt ::= "throw" Expression
```

Executing this statement, whose value is null, throws the value of `Expression`. The value of `Expression` shall be an exception.

Process statements

A process statement is one of the following:

```
ProcessStmt ::= OwnStmt
              | RestrictStmt
              | UseStmt
```

The `own` statement

The `own` statement is written as follows:

```
OwnStmt ::= "own" Block
```

Executing this statement, whose value is null, notes the current owner, *P*; instates the owner of the current object as the current owner; executes `Block`; and reinstates *P* as the current owner, whether or not `Block` throws an exception.

The `restrict` statement

The `restrict` statement is written as follows:

```
RestrictStmt ::= "restrict" Expression Block [CatchPhrase]
```

Executing this statement, whose value is null, instates as a temporary permit the value of `Expression`, which shall be a permit; executes `Block`; and deinstates the temporary permit even if `Block` throws an exception.

The statement can handle a permit violation as follows. If `Block` violates the current permit (which reflects the temporary permit), it necessarily throws Permit Violated. If `Catchphrase` occurs, the statement handles the exception by executing `Catchphrase`. The `Type` in `CatchPhrase` shall be one whose base class is Permit Violated or an interface superclass thereof, whose *isSubclassOK* attribute is *true*, and whose *isNilOK* attribute is *false*.

The `use` statement

The `use` statement is written as follows:

```
UseStmt ::= "use" Expression1
            ["shared"] [Condition] Block1 [TimeOut]

Condition ::= "when" Expression2

TimeOut ::= "after" Expression3 Block2
```

Executing this statement, whose value is null, acquires use of the value of `Expression`$_1$, which shall be a resource; executes `Block`$_1$, a conditional critical region; and relinquishes use of the resource even if `Block`$_1$ throws an exception. The resource's use is acquired only on the following terms:

- If "`shared`" occurs, the use is shared. Otherwise the use is exclusive.
- If `Condition` occurs, the resource's condition must match an item of the value of `Expression`$_2$, which shall be a set of identifiers matching items of the resource's `conditions` attribute. If the resource has only the one, distinguished, undefined condition, `Expression`$_2$ shall not occur.
- If `TimeOut` occurs in the statement and the statement would have to wait longer than the number of seconds that is the value of `Expression`$_3$, which shall be an integer, for the two foregoing terms to be met, the statement executes `Block`$_2$ rather than `Block`$_1$.

Expressions

Expressions fall conveniently into the following categories. Basic expressions get and set local variables and properties; get global variables and classes; and assert objects' types. Operator expressions offer the customary syntax for requesting arithmetic and logical operations and have their results as values. General operation expressions request arbitrary operations, including getters and setters, and have their results as values. Special operation expressions provide concise syntax for requesting commonly used operations and have their results as values. Literal expressions have primitives and selected collections as values. Escapes provide direct access to the engine:

```
Expression ::=  Assignment | Operand

   Operand ::=  Operator | Object

    Object ::=  Module
             |  ModuleItem
             |  "(" Statement ")"
             |  Access
             |  Assertion
             |  GeneralRequest
             |  SpecialRequest
             |  Literal
             |  Escape
```

Note Even though they're described in other chapters of this manual, modules, module items, and (parenthesized) statements qualify as expressions.

Basic expressions

A basic expression is one of the following.

Accessing an object

An object is accessed as follows:

```
    Access ::=  "*" | ID
```

The expression's value is as follows. If "`*`" occurs, the value is an unprotected reference to the current object. Otherwise the value is the object `ID` denotes.

`ID` shall denote an object described by the following table. Each row describes zero or more objects to which identifiers are bound. The rows are arranged in order of decreasing precedence. Of all the objects the table describes, `ID` denotes the one of highest precedence with that identifier.

Category	*Objects in category*
Global variable	The global variables.
Local variable	The local variables declared in the visible variable declaration segments of the enclosing blocks—from the block that is most tightly enclosing to the block that is the current method.
Property	The properties of the current object the current class defines.

Classes	The classes that ID may denote (see “Specifying a class”).

Note: The arguments of the current operation to which the operation’s declaration assigns identifiers are accessible as local variables (see “Defining an operation’s method”).

Assigning an object

An object is assigned as follows:

```
Assignment ::=  ID“=” Expression
              | SetterRequest
              | SetRequest
```

The first form of assignment is defined as follows. The object that ID denotes (see “Accessing an object”) is replaced with the value of Expression. ID shall denote a local variable or a property, and the value of Expression shall not be null. The value of Assignment is the value of Expression.

The second and third forms of assignment are defined in “General operation expressions” and “Special operation expressions”, respectively.

Asserting an object’s type

An object’s type is asserted as follows:

```
Assertion ::=  Object “@” [ClassID]
```

The expression’s value is Object’s which shall not be null. However, the compiler (but not the engine) considers the expression’s value to be an interface member of an *asserted class*. If ClassID occurs, the asserted class is the class it denotes. Otherwise it’s inferred from the expression’s context.

The asserted class shall relate to the *actual class*, that of which the compiler considers the value of Object to be an interface member, in one of two ways:

- The asserted class shall be the actual class or an interface subclass of it.
- The asserted and actual classes shall be derived from the same class family. Each parameter used in one derivation shall be the corresponding parameter used in the other or an interface subclass thereof.

Notes.

- For example, the first point above allows here to be considered an interface member of Meeting Place rather than Place.
- For example, the second point allows an interface member of List[Integer, Equal] to be considered an interface member of List[Number, Equal], but also vice versa.

Operator expressions

A prefix or infix operator is applied as follows:

```
Operator ::=  PrefixOperator | InfixOperator
```

Note: An operator isn’t to be confused with an operation. An operator is mapped during compilation into one or more operations performed during execution.

Applying a prefix operator

A prefix operator is applied as follows:

```
PrefixOperator ::= «prefix» Operand
```

This metarule, which represents all prefix operators, abbreviates a family of rules which introduce operator symbols and a nonterminal syntactic class. The form "*•x*" abbreviates the form "*x*.*id* ()". *•* is the prefix operator, *x* the `Operand`, and *id* the identifier of an operation which *•* determines.

The expression's value is the result of an operation in the following table—the operation that the table associates with the prefix operator. `Operand`'s value shall be an interface member of a class that defines that operation.

Operator	*Definition*
`!x`	`x.not()`
`-x`	`x.negate()`

The prefix operators are left-associative and higher in precedence than the infix operators (see "Applying an infix operator").

Notes.

- The direct use of the predefined *operations* in the table is denigrated.
- The table operations may be either the predefined ones or user-defined operations with the same identifiers and suitable arguments and results.

Applying an infix operator

An infix operator is applied as follows:

```
InfixOperator ::= Operand1 «infix» Operand2
```

This metarule, which represents all infix operators, abbreviates a family of rules which introduce operator symbols and a corresponding hierarchy of nonterminal syntactic classes, one per precedence level (see below). In the simplest cases, the form "*x•y*" abbreviates the form "*x*.*id* (*y*)". *•* is the infix operator, *x* is $\texttt{Operand}_1$, *y* is $\texttt{Operand}_2$, and *id* is the identifier of an operation which *•* determines. In more complex cases, several operations are involved, the result of one serving as an argument for the next.

The expression's value is the result of an operation in the following table—the operation that the table associates with the infix operator. Where the table associates several operations with an infix operator (as it does, for example, with "<"), the expression's value is the result of the operation performed last. The value of $\texttt{Operand}_1$ shall be an interface member of a class that defines that operation. The value of $\texttt{Operand}_2$ shall not be null.

Operator	*Definition*
`x*y`	`x.multiply(y)`
`x/y`	`x.divide(y)`
`x+y`	`x.sum(y)`
`x-y`	`x.difference(y)`

<table>
<tr><td><code>x is C</code></td><td><code>x.isMember(C)</code></td></tr>
<tr><td><code>x is C!</code></td><td><code>x.isInstance(C)</code></td></tr>
<tr><td><code>x<y</code></td><td><code>x.compare(y) == before</code></td></tr>
<tr><td><code>x>y</code></td><td><code>x.compare(y) == after</code></td></tr>
<tr><td><code>x<=y</code></td><td><code>!(x>y)</code></td></tr>
<tr><td><code>x>=y</code></td><td><code>!(x<y)</code></td></tr>
<tr><td><code>x==y</code></td><td><code>x.isEqual(y)</code></td></tr>
<tr><td><code>x!=y</code></td><td><code>!(x==y)</code></td></tr>
<tr><td><code>x&&y</code></td><td><code>if x {y} else {false}</code> (that is, conditional and)</td></tr>
<tr><td><code>x||y</code></td><td><code>if x {true} else {y}</code> (that is, conditional or)</td></tr>
</table>

The sections of the table are arranged in order of decreasing operator precedence. The operators within a section all have the same precedence.

The compiler treats most operations that Number defines as if Integer and Real defined them, and specializes accordingly the types of their results and, for Integer, their arguments. Thus, for example, if the `divide` operation's intended responder is an integer but its argument is a real, the programmer must convert the integer to a real before requesting the operation of it.

Note. The infix operator "`is`" is defined by two rows of the table, but could be defined instead by a single row as "`x is T`", where `T` is a type whose *isNilOK* boolean is *false* (and whose *isSubclassOK* boolean is either *true* or *false*).

Notes.

- The direct use of the predefined *operations* in the table is denigrated.
- The table operations may be either the predefined ones or user-defined operations with the same identifiers and suitable arguments and results.

General operation expressions

An individual operation or a cascade of operations is requested as follows:

```
GeneralRequest ::=  CascadeRequest
                 |  OperationRequest
```

Specifying the responder

The responder is specified as follows:

```
     Responder ::=  Object | Self "::" [ClassID]
          Self ::=  "*" | "self"
```

The responder and how it performs an operation are as follows:

- *Normal.* If `Object` occurs, its value, which shall not be null, performs the operation normally (that is, the method search begins with the responder's class). If the operation is private, `Object` shall be `Self`.
- *Escalated.* If `Object` doesn't occur, the current object escalates the operation. If `ClassID` occurs, the method search begins with the class it denotes, which shall be the current class or an immediate implementation superclass thereof. Otherwise the search begins with the class among the latter that is *last* in canonical order.

Note. `Responder` appears directly in `OperationRequest` and indirectly in `CascadeRequest`.

Specifying the arguments

The arguments are specified as follows:

```
ArgumentList ::= "(" [Arguments] ")"

   Arguments ::= FixedGroups
               | VaryingGroup
               | FixedGroups "," VaryingGroup
```

The arguments fixed in number are the values of the `Expressions` in `FixedGroups`, none of which shall be null. Each actual argument shall satisfy the type associated with the like-positioned formal argument. The actual and formal arguments shall agree in number. However, the last one or more arguments fixed in number may be unspecified and so taken to be nil:

```
 FixedGroups ::= Expressions

 Expressions ::= Expression | Expressions "," Expression
```

The arguments varying in number are the items of the value of the `Expression` in `VaryingGroup`, which shall be a list. Each actual argument shall satisfy the type associated with the formal argument:

```
VaryingGroup ::= Expression "..."
```

Notes.

- For the constructor, the rule above that lets arguments be left unspecified has in view the arguments as declared by the current class's immediate implementation superclasses *collectively* (see "Deciding the constructor's actual arguments").
- `ArgumentList` appears directly in `OperationRequest` and `CascadeRequest`.

Requesting a cascade of operations

A cascade of operations, but not setters, is requested as follows:

```
CascadeRequest ::= OperationRequest Others

        Others ::= Other | Others Other

         Other ::= "&." ID
                 | "&." ID ArgumentList
```

The expression's value is the cascade's result. If the cascade has no result, the expression's value is null. The cascade's responder and the cascade's first

operation and its arguments are given by `OperationRequest`. Every other operation and its arguments are given by an `Other` in the same way.

The syntax rules for `CascadeRequest`, more complicated than indicated above, handle with a minimum of parentheses—yet without ambiguity—the intermixing of general requests for operations, special requests for the `get` operation, and type assertions. They recognize "`e@ID[ ... ]`" as an assertion involving a class family, "`(e@ID)[ ... ]`" as a request for the `get` operation.

Notes.

- The following exemplifies a cascade of getters. "$o.ID_1$ $\&.ID_2$... $\&.ID_n$" abbreviates "$o.ID_1$; $o.ID_2$;... $o.ID_n$" but evaluates `o` only once.
- The following exemplifies a cascade of operations other than getters. "$o.ID_1(e_1)$ $\&.ID_2(e_2)$... $\&.ID_n(e_n)$" abbreviates "$o.ID_1(e_1)$; $o.ID_2(e_2)$;... $o.ID_n(e_n)$" but evaluates `o` only once.

Requesting an operation

An operation, but not a setter, is requested as follows:

```
OperationRequest ::=     Responder "." ID
                       | Responder "." ID ArgumentList
```

The expression's value is the operation's result. If the operation has no result, the expression's value is null. The responder and how it performs the operation are specified by `Responder`. The operation's arguments are specified by `ArgumentList`. If `ArgumentList` occurs, the operation is the one `ID` denotes. Otherwise it's the getter of the attribute `ID` that denotes. In either case, the operation shall be one that the responder performs.

Requesting a setter

A setter is requested as follows:

```
SetterRequest ::=  GeneralRequest "=" Expression
```

The expression's value is `Expression`'s, which shall not be null. The attribute that `GeneralRequest` denotes is set to that value.

The primary use of the `GeneralRequest` rule is to request an operation *other* than a setter (as described in the preceding subsections). In the present context, the requested operation, or the last operation in the requested cascade, shall be an attribute's getter. However, the `GeneralRequest` is understood to request the setter, rather than the getter, for that attribute.

Notes `SetterRequest` is an `Assignment`, not a `GeneralRequest`. Nevertheless it's presented in this section because of its affinity with `GeneralRequest`.

Special operation expressions

Any of certain commonly used operations is concisely requested as follows:

```
SpecialRequest ::=  NewRequest
                  | GetRequest
                  | GetterRequest
                  | Escalation
```

The expression's value is the operation's result. If the operation has no result, the expression's value is null.

Requesting the `new` operation

The predefined `new` operation can be requested as follows:

```
NewRequest ::= ClassID ArgumentList
```

`NewRequest` abbreviates `OperationRequest`. The `new` operation is requested, not escalated, with the arguments that `ArgumentList` specifies. The responder is the class that `ClassID` denotes, which shall be concrete.

To find the class, the engine follows the class search algorithm using packages that the current object's owner offers privately (see the class manual). If the algorithm fails, the engine throws Class Unavailable.

Note. Requesting the `new` operation using `OperationRequest`, rather than `NewRequest`, is denigrated because the constructor's arguments are not checked.

Requesting a `get` operation

A `get` operation can be requested as follows:

```
GetRequest ::= Responder "[" Expressions "]"
```

`GetRequest` abbreviates `OperationRequest`. The `get` operation is requested or escalated—whichever `Responder` requires—with no arguments varying in number and with the values of `Expressions`, none of which shall be null, as the arguments fixed in number. However, the last one or more arguments fixed in number may be unspecified and so taken to be nil. The responder, which `Responder` specifies, shall be an interface member of a class that defines a `get` operation with arguments as above.

Notes.

- "o[e_1,... e_n]" abbreviates " o.get(e_1,... e_n)".
- Dictionaries, lists, and package processes qualify as responders.

Requesting a `set` operation

A `set` operation can be requested as follows:

```
SetRequest ::= Responder "[" Expressions "]" "=" Expression
```

`SetRequest` abbreviates `OperationRequest`. The `set` operation is requested or escalated—whichever `Responder` requires—with no arguments varying in number and with the values of `Expressions` and `Expression`, none of which shall be null, as the arguments fixed in number. However, the last one or more arguments fixed in number may be unspecified and so taken to be nil. The responder, which `Responder` specifies, shall be an interface member of a class that defines a `set` operation with arguments as above.

Notes.

- "o[e_1,... e_n] = e_{n+1}" abbreviates " o.set(e_1,... e_n, e_{n+1})".
- Dictionaries and lists qualify as responders.
- `SetRequest` is an `Assignment`, not a `SpecialRequest`. Nevertheless it's presented in this section because of its affinity with `GetRequest`.

Requesting a getter

A getter can be requested if the responder isn't nil as follows:

```
GetterRequest ::=  Responder "?." ID
```

`GetterRequest` abbreviates and extends `OperationRequest`. The getter of the attribute `ID` denotes is requested or escalated—whichever `Responder` requires—with no arguments. The responder is specified by `Responder`. If the responder is nil, the getter isn't requested; nil supplants its result.

Escalating the current operation

The current operation can be escalated as follows:

```
Escalation ::=  "^" [ArgumentList]
```

`Escalation` abbreviates `OperationRequest`. The current object escalates the current operation so as to begin the method search with the current class's immediate implementation superclass first in canonical order. If `ArgumentList` occurs, it specifies the arguments. Otherwise a telescript escape (see "Escape expressions") must have left them on top of the current stack.

In this version of the language, if a mix-in among the class's immediate implementation superclasses redefines the constructor to have arguments varying in number, the class's method shall supply the arguments using a telescript escape and shall escalate the constructor as described here.

Note Arguments are pushed onto the current stack using an `Escape`. The compiler can't check the types of arguments supplied in this way. The use of `Escape` and thus of the form of `Escalation` that requires it is denigrated.

Literal expressions

A primitive or collection is denoted as follows. Of the language's wide variety of collections, only bit strings, octet strings, and strings are literally denoted:

```
Literal ::=  Bit
          |  BitString
          |  Boolean
          |  Character
          |  Identifier
          |  Integer
          |  Nil
          |  Octet
          |  OctetString
          |  Real
          |  String
```

The expression's value is the primitive or collection it denotes. The reference to that primitive or collection is a protected reference.

Denoting a bit

A bit is denoted as follows:

```
Bit ::=  BIT
```

A `BIT`—for example, `%1`—is "`%`" followed by either "`0`" or "`1`". "`%0`" denotes zero. "`%1`" denotes one.

Denoting a bit string

A bit string is denoted as follows:

```
BitString ::= BITSTRING
```

A BITSTRING—for example, `%"0111000101"`—is "`%`" followed by zero or more characters enclosed in "`"`". Each character encodes a bit as it would in a BIT and is similarly constrained. The expression denotes a bit string whose items are the encoded bits; the ith character encodes the bit at position *i*.

Denoting a boolean

A boolean is denoted as follows:

```
Boolean ::= "true" | "false"
```

"`true`" denotes *true*. "`false`" denotes *false*.

Denoting a character

A character is denoted as follows:

```
Character ::= CHARACTER
```

A CHARACTER—for example, `'a'`—is either a single character or an escape sequence enclosed, in either case, in "`'`". If a single character occurs, the expression denotes that character. If an escape sequence occurs, the expression denotes the character that the escape sequence denotes.

Denoting an identifier

An identifier is denoted as follows:

```
Identifier ::= ATOM
```

An ATOM—for example, `'copy`—is "`'`" followed by the text of an identifier. The identifier is mentioned, not used, during the program's compilation.

In contrast to an ATOM, an ID—for example, `copy`—is the text of an identifier. The identifier is used, not mentioned, during the program's compilation.

Denoting an integer

An integer is denoted as follows:

```
Integer ::= INTEGER
```

An INTEGER—for example, `123`—is a sequence of one or more characters, each in "0" through "9". The expression denotes the non-negative integer that the digits encode. (The negate operator yields negative integers.)

Denoting nil

Nil is denoted as follows:

```
Nil ::= "nil"
```

"`nil`" denotes nil.

Denoting an octet

An octet is denoted as follows:

```
Octet ::= OCTET
```

An `OCTET`—for example, `$ff`—is "`$`" followed by a character pair. Each character in the pair is in "0" through "9", "A" through "F", or "a" through "f". The expression denotes the octet whose Bits 4-7 the first character encodes and whose Bits 0-3 the second character encodes.

Denoting an octet string

An octet string is denoted as follows:

```
OctetString ::= OCTETSTRING
```

An `OCTETSTRING`—for example, `$"6789ab"`—is "`$`" and zero or more character pairs enclosed in "`""`". Each pair encodes an octet as it would in an `OCTET` and is similarly constrained. The expression denotes an octet string whose items are the encoded octets; the ith pair encodes the octet at position *i*.

Denoting a real

A real is denoted as follows:

```
Real ::= REAL
```

A `REAL`—for example, `314.159E-2` or `1e6`—encodes a mantissa, *m*, and perhaps an exponent, *e*, which if not encoded is 0. The expression denotes the non-negative real, $m \times 10^{e}$. (The negate operator yields negative reals.)

m is encoded as one or more characters, each in "`0`" through "`9`", optionally preceded or separated by "`.`". (If "`.`" occurs, at least one character follows it.) If "`.`" occurs, the encoding of *e* is optional. Otherwise it's mandatory.

e is encoded as "`e`" or "`E`", optionally followed by "`+`" or "`-`", mandatorily followed by one or more characters, each in "0" through "9". If "`-`" doesn't occur, *e* is the non-negative integer that the characters in "0" through "9" encode. Otherwise *e* is the arithmetic negation thereof.

Denoting a string

A string is denoted as follows:

```
String ::= STRING
```

A `STRING`—for example, `"abc"`—is zero or more characters and escape sequences enclosed in "`""`". Each character or escape sequence denotes a character as it would in a `CHARACTER` and is similarly constrained. The expression denotes a string whose items are the denoted characters. The ith character or escape sequence denotes the character at position *i*.

Escape expressions

A *telescript escape* is denoted as follows:

```
Escape ::= ESCAPE
```

An `ESCAPE` is zero or more characters surrounded by "<<" and ">>". The characters encode as a character telescript zero or more items of an object program. The compiler includes the items in the object program it produces.

The expression's zero or more *values* are the objects by which execution of the above object program items increases the length of the current stack. If execution decreases the current stack's length, the expression's value is null.

Among the roles that an `Escape` can play are those in the following table. In any of these roles, an `Escape` shall have the indicated objects as its values. In any other role, an `Escape` shall satisfy requirements that are beyond the scope of this manual (and that depend upon the compiler's implementation).

Role	*Values*
`ModuleItem`	The definition of zero or more classes.
`Method`	The definition of a method whose performance changes the current stack as the operation's signature requires.
An argument	One object.
The right-hand side of an assignment	One object.
A statement	No objects.

Notes.

- In an ESCAPE a reserved word is used as an ID without a prefix.
- The use of `Escape` is denigrated.

Programs

A Telescript program has two forms, source and object. Both source and object programs, as well as programs in the abstract, are described below.

Abstract programs

A Telescript *abstract program*, a list of statements, is *executed* by executing the statements from left to right, discarding the values of all but the last, and taking the value of the last statement as the abstract program's *value*.

An abstract program is defined as follows:

```
       Program ::=  Statements

    Statements ::=  Statement
                 |  Statements ";" Statement
                 |  Statements ";"
```

Notes.

- The value of the typical abstract program is a module.
- Executing an abstract program whose value is a module executes none of the statements in the methods defined by the classes in the module. Execution of those statements is deferred (see "Defining an operation's method").
- One may view the "`;`" as either a separator or a terminator. It is mandatory between `Statements` but is optional after the last `Statement`.

Source programs

A Telescript *source program* is a string (of Unicode characters) that encodes a Telescript abstract program by following the syntactic and semantic rules of this manual. The string itself is encoded by following the syntactic and semantic rules of *File System Safe UCS Transformation Format (FSS_UTF)*.

A source program can include preprocessing directives, breaks, comments, escape sequences recognized as single characters, and reserved words recognized as identifiers. These possibilities are defined below.

Including a preprocessing directive

A source program is compatible with the C preprocessor.

Note. Thus one can include text and define abbreviations using the familiar `#include` and `#define` directives, respectively.

Including a break

A source program can include breaks. A *break* is one or more characters that either separate program tokens or format the source program without altering the abstract program. Each break character is "SPACE", "HT", or "LF".

A break can occur before the first program token, after the last, or between adjacent tokens. A break *must* occur between tokens that consist only of characters in "`0`" through "`9`", "`A`" through "`Z`", and "`a`" through "`z`".

Including a comment

A source program can include comments. A *comment* is a sequence of characters that annotates the abstract program without altering it.

A comment takes one of the following two forms:

1. "/*", one or more characters excluding "*/", and "*/".
2. "//", one or more characters excluding "LF", and "LF".

One or more comments can occur before the first program token, after the last, or between adjacent tokens. (None, of course, are required.)

Notes.

- A comment of the first form can include "/*" or "//" without special meaning and thus can include a comment of the second form. A comment of the second form can include "/*", "*/", or "//" without special meaning and thus can include a comment of the first form. A comment of the second form can't be followed on the same line by other tokens; the comment would include them.
- The syntax rules for comments are those of C++.

Including an escape sequence

A source program can include escape sequences in its denotations of strings and characters. An *escape sequence* is two or more characters of a STRING (see "Denoting a string") that denote a single character, or two or more characters of a CHARACTER (see "Denoting a character") that denote the sole character.

The escape sequences are defined by the following two tables. The first columns list the escape sequences, the second the characters they denote. The characters in the first table shall occur *only* in escape sequences.

Escape sequence	*Character*
"\a" or "\A"	"BEL"
"\b" or "\B"	"BS"
"\f" or "\F"	"FF"
"\n" or "\N"	"LF"
"\r" or "\R"	"CR"
"\t" or "\T"	"HT"
"\v" or "\V"	"VT"
"\\"	"\"
"\?"	"?"
"\'"	"'"
"\""	"""

Escape sequence	*Character*

"\x*h...h*" or "\X*h...h*"	The character whose Unicode value is *h...h*$_{16}$. "*h...h*" stands for one or more characters—each in "0" through "9", "A" through "F", or "a" through "f"—which express an unsigned integer in hexadecimal.
"*o...o*"	The character whose Unicode value is *o...o*$_8$. "*o...o*" stands for one, two, or three characters—each in "0" through "7"—which express an unsigned integer in octal.
"*c*"	The character *c*. "*c*" stands for any character other than "x", "X", "0" through "7", and those characters that occur in the escape sequences in the first column of the previous table.

Note. The escape sequences are those of ANSI C.

Including a reserved word

A source program can include reserved words in a way that allows the compiler to recognize them as identifiers rather than keywords. A *reserved word* is a terminal that must be prefixed with "_" to be recognized as an ID.

The reserved words are the following:

abstract	imports	place	sponsor
after	in	private	sponsored
break	instance	process	throw
catch	interface	property	throws
class	is	protected	to
client	loop	public	true
continue	magiccap	readonly	try
copied	mixin	ref	unprotected
do	module	repeat	use
else	nil	restrict	when
false	op	return	while
for	own	sealed	with
here	owned	self	
if	owner	shared	

Notes.

- For example, "_abstract" is the ID whose text is "abstract".
- The identifiers of predefined classes and their features aren't, in general, reserved words. Such identifiers are reserved words only as listed above.

Including a character category

The following table lists and defines the categories of Unicode character to which the language and the predefined classes collectively refer.

Character category	*Definition*
alphabetic characters	The characters that are neither control, decimal digit, nonspacing mark, private use, punctuation, space, nor special characters nor noncharacters

ASCII characters	Section 3.1 of the Unicode specification
control characters	Section 2.4 of the Unicode specification
decimal digit characters	Section 4.1 of the Unicode specification
lowercase characters	Section 5.5 of the Unicode specification
noncharacters	The characters with Unicode values U+FFFE and U+FFFF (see p. 123 of the Unicode specification)
nonspacing mark characters	Section 4.5 of the Unicode specification
private use characters	Section 3.5 of the Unicode specification
punctuation characters	• “ !” through “ /” • “ :” through “ @” • “ [” through “ ^” • “ `” • “ {” through “ ~” • “ ¡” through “ ¿” • “ ×” and “÷” • U+2000 through U+2FFF
space characters	Section 4.2 of the Unicode specification
special characters	Section 3.6 of the Unicode specification
uppercase characters	Section 5.5 of the Unicode specification

Including a character

The following table lists and defines the individual Unicode characters to which the language and the predefined classes collectively refer.

The manuals show the typical images of various Unicode characters in quotation marks (for example, “ *”). The table lists the Unicode names and the Unicode values in hexadecimal of these characters.

Unicode value	*Typical image*	*Unicode name*
0000	NUL	NULL
0007	BEL	BELL
0008	BS	BACKSPACE
0009	HT	HORIZONTAL TABULATION
000A	LF	LINE FEED
000B	VT	VERTICAL TABULATION
000C	FF	FORM FEED
000D	CR	CARRIAGE RETURN
0020	SPACE	SPACE
0021	!	EXCLAMATION MARK
0022	"	QUOTATION MARK

0023	#	NUMBER SIGN
0024	$	DOLLAR SIGN
0025	%	PERCENT SIGN
0026	&	AMPERSAND
0027	'	APOSTROPHE-QUOTE
0028	(	OPENING PARENTHESIS
0029	)	CLOSING PARENTHESIS
002A	*	ASTERISK
002B	+	PLUS SIGN
002C	,	COMMA
002D	-	HYPHEN-MINUS
002E	.	PERIOD
002F	/	SLASH
0030...0039	0...9	DIGIT ZERO... DIGIT NINE
003A	:	COLON
003B	;	SEMICOLON
003C	<	LESS-THAN SIGN
003D	=	EQUALS SIGN
003E	>	GREATER-THAN SIGN
003F	?	QUESTION MARK
0040	@	COMMERCIAL AT
0041...005A	A...Z	LATIN CAPITAL LETTER A... LATIN CAPITAL LETTER Z
005B	[	OPENING SQUARE BRACKET
005C	\	BACKSLASH
005D	]	CLOSING SQUARE BRACKET
005E	^	SPACING CIRCUMFLEX
005F	_	SPACING UNDERSCORE
0060	`	SPACING GRAVE
0061...	a	LATIN SMALL LETTER A
007A	z	LATIN SMALL LETTER Z

007B	{	OPENING CURLY BRACKET
007C	|	VERTICAL BAR
007D	}	CLOSING CURLY BRACKET
007E	~	TILDE
00A1	¡	INVERTED EXCLAMATION MARK
00BF	¿	INVERTED QUESTION MARK
00D7	×	MULTIPLICATION SIGN
00DF	β	LATIN SMALL LETTER SHARP S
00F7	÷	DIVISION SIGN

Object programs

A Telescript *object program* is the result of compiling a Telescript source program. While its detailed nature is beyond the manual's scope, an object program takes either of two equivalent forms. A *binary telescript* encodes an object program as an octet string. A *character telescript* encodes it as a string.

The predefined classes define operations for converting certain objects to elements of binary and character telescripts and for converting such elements to such objects. These operations are summarized in the following table.

	To binary telescript *From binary telescript*	*To character telescript* *From character telescript*
From integer *To integer*	— —	• Integer's `asString` • String's `asInteger`
From real *To real*	— —	• Real's `asString` • String's `asReal`
From string *To string*	• String's `asOctetString` • Octet String's `asString`	— —

Note: The use of the operations in the table is denigrated, at least to the extent that such use assumes knowledge of the formats of the telescripts involved.

Part Three— Predefined Class Concepts

This part of the manual defines the major concepts—in particular, the agent and place abstractions—that underlie the predefined classes.

Chapters are devoted to the following topics:

1. places, which provide venues for agents and lets them interact.
2. agents, which can transport themselves from one place to another.
3. processes, which include places and agents.
4. permits, which determine the capabilities of processes.
5. patterns, tools for analyzing and modifying strings.
6. calendar times, tools for analyzing and modifying times.

Places

A Telescript *place* is a process (see “Processes”) that provides a venue for other processes (its occupants) and that may allow them to interact. This chapter discusses how places relate to one another and to other processes.

Organizing the telesphere

The *telesphere* is the universe of places. The telesphere is divided, for operational reasons, among one or more regions. The places of a region are divided, for computational reasons, among one or more engines.

Organizing a region

A *region* consists of the places that are sustained by the engines that are operated by a particular person or organization, the region’s *authority*.

Notes.

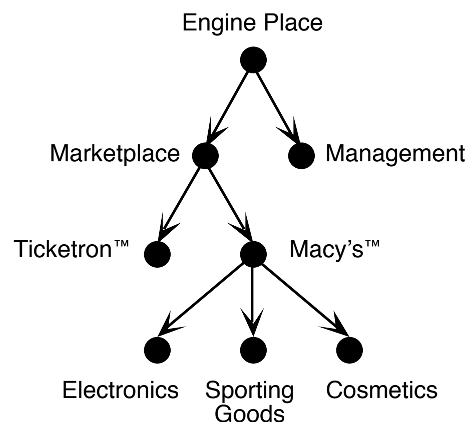
- A region can be as large as a public value-added network (for example, the AT&T PersonaLink Service). Such a region, whose authority is that of the service operator (in this case, AT&T), might contain many engines.
- A region can be as small as a user’s personal intelligent communicator. Such a region, whose authority is that of the user, might contain a single engine.

Organizing an engine

An *engine* consists of one *engine place* and zero or more *virtual places*. A virtual place occupies another place; an engine place doesn’t.

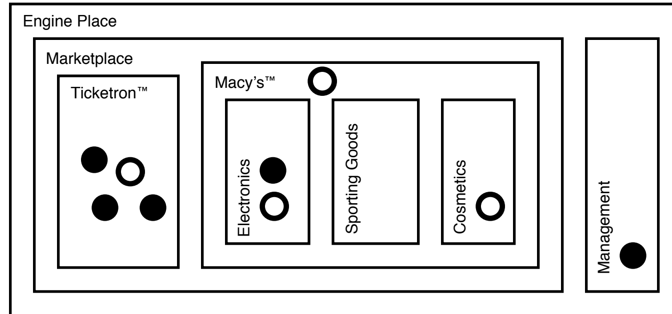
One place may *occupy* another. The first of these two places is an *immediate subplace* of the second; the second is the *immediate superplace* of the first. If the second place occupies a third place, the first and second places are *subplaces* of the third; the third is a *superplace* of the first and second.

A tree, the *place hierarchy*, illustrated, defines the occupancy relationships among an engine place and its subplaces. The tree’s nodes represent places, and the arc between two nodes represents the occupancy relationship between those two places. The source node represents the immediate superplace of the place that the destination node represents. The tree’s root represents the engine place.



Not all occupants of a place are necessarily places themselves. Zero or more agents *occupy* each place as well.

A place can be visualized as “containing” its occupants. In the tiny electronic marketplace depicted in the next illustration, the outer rectangle represents an engine place, while other rectangles represent virtual places in the role of merchant. In the same illustration, the white circles represent agents in the role of seller, while the black circles represent agents in the role of buyer.



Addressing a place

All places have distinctive names (see "Naming a process"). *Advertised places* also have distinctive addresses, which locate them in the telesphere. Engine places are advertised. A virtual place is advertised at its region's discretion. An unadvertised place has the address that its immediate superplace has.

A *teleaddress* purports to denote one advertised place or all advertised places in a given region. In the first case, the teleaddress is an *assigned teleaddress*.

Specifying a region

A teleaddress specifies a region with an octet string—the octet string with which a telename specifies the region's authority (see "Naming a process").

Specifying a place

A teleaddress specifies an advertised place with an octet string. Chosen using an algorithm that is region-specific, the octet string distinguishes the advertised place from other advertised places in the region.

Specifying routing advice

A teleaddress offers routing advice. *Routing advice* specifies zero or more transit regions in a preferred order. Whenever an agent indicates its travel destination using the teleaddress (see "Constructing a ticket"), the `go` or `send` operation considers routing the agent through one of those regions.

A teleaddress specifies a transit region with the octet string with which a telename specifies the transit region's authority (see "Naming a process").

The assigned teleaddresses of all places in an engine give the same advice.

Managing occupants

A place is *occupied* by other processes—agents, places, or both. This section describes how occupants come and go and how the place keeps track of them.

Entering a place

A process comes to occupy a place by *entering* it. A process can enter a place by being constructed there either by the place itself or by an agent already among its occupants. Also, an agent can enter a place by taking a trip there.

The engine mediates the entry of a process to a place. The engine requests the `entering` operation of the place. The operation's arguments identify the

process. If the operation succeeds, entry occurs. Otherwise entry fails and the engine hides or reveals the place's existence—whichever the place requests.

The engine doesn't serialize the entry of processes to a place. Even if a place is already performing the `entering` operation, the engine isn't deterred from requesting the operation again on behalf of other processes.

Note: The place can provide its own serialization (for example, using a resource).

Exiting a place

A process comes to no longer occupy a place by *exiting* it. A process can exit a place either by completing performance of the `live` operation or by being terminated. Also, an agent can exit a place by starting a trip.

The engine mediates the exit of a process from a place. The engine sends a signal, a member of Exit Event, whose source and intended destination alike are that place. The place receives the signal when and if it chooses.

Note: The engine requests the `entering` operation and sends an exit event even if the origin and destination of an agent's trip are one and the same place.

Keeping track of occupants

A place designates an object as its *record* of a particular occupant. The place supplies the object to the engine as the result of the `entering` operation. The engine supplies the object to the place as an attribute of the exit event.

Note: From time to time a place may modify its record of an occupant so that the object reflects the state of the place's interaction with that occupant.

Agents

A Telescript *agent* is a process that can transport itself from one place in the telesphere to another. Often an agent makes such a trip to meet with another agent. This chapter discusses how agents travel and meet.

Traveling to other places

An agent can take a *trip* from one place, its *origin*, to another or the same place, its *destination*. If the trip succeeds, the agent is left at its destination. If the trip fails, the agent is left either at its origin or in a third place. The third place is either in the same region as the origin or in a different region.

In principle, but rarely in practice, a trip can result in an agent's duplication. A *duplicate* of an agent is an inadvertent copy that arises from travel. If its `isPossibleDuplicate` attribute is *true*, an agent might be a duplicate.

Note. An agent's trip can take a long time because it may involve transporting the agent by means of physical, not just logical, communication media.

Constructing a ticket

A *ticket* defines a trip that an agent might take. The ticket's main purpose is to identify the agent's destination.

A ticket's attributes provide information drawn from the following table. The first column identifies the attributes; the second describes them.

Identifier	*Attribute*
`destinationAddress`	The teleaddress required of the destination.
`destinationClass`	The interface member class required of the destination.
`destinationName`	The telename required of the destination.
`destinationPermit`	The local and regional permits desired at the destination.
`wayOut`	The required way, and thus perhaps the required means, of travel from the agent's origin toward its destination (see "Selecting a route").
`desiredWait`	The desired maximum duration of the trip.
`maximumWait`	The required maximum duration of the trip, after which the ticket *expires*. (The same effect can be achieved using a temporary permit.)
`notes`	The agent's notes concerning the trip.

Notes.

- The destination is the one place with a given name; any place of a given authority; any place at a given address; any place in a given region; or any place of a given class. These constraints can be combined.

- Defining the desired maximum duration of a trip helps the telesphere decide how to distribute its communication resources among agents in transit.

Satisfying a ticket

A ticket constrains an agent's destination, but not necessarily to a single place. A ticket is satisfied in principle by any place with the name, address, and class that the ticket specifies. The ticket is *satisfied* in practice by the one place (if any) that agrees to the agent's entry before the ticket expires.

The language doesn't require the engine to wait for a ticket to expire before ending its search for places that satisfy the ticket in principle. Thus a place constructed after the ticket is presented but before it expires may not qualify.

The engine asks places that satisfy the ticket in principle to perform the `entering` operation (see "Entering a place"). The engine approaches these places one after another, rather than in parallel; approaches them in an undefined order; and approaches each place at most once.

Receiving a ticket stub

An agent receives a ticket stub when it completes a trip. The main purpose of the *ticket stub* is to provide the agent with a way back to its origin.

A ticket stub's attributes provide information drawn from the following table. The first column identifies the attributes; the second describes them.

Identifier	*Attribute*
`wayBack`	A way that the agent can take to return to its origin. (This can involve an existing connection means.)
`isConstrained`	*True* if the trip succeeded, but resulted in a local or regional permit that is more constraining than the one that the agent requested by means of its ticket.
`notes`	The agent's notes concerning the trip, as supplied originally in the agent's ticket.

Going to another place

An agent, using the `go` operation, can travel to the place specified by a ticket.

Note: The language's `go` and `send` operations are its most distinctive and powerful aspect. In most network architectures, processes are stationary and communicate by exchanging messages. In this one, some processes are stationary but others are mobile. The former communicate by means of the latter.

Sending clones to other places

An agent, using the `send` operation, can cause clones to be constructed and to travel concurrently to the places specified by one or more tickets.

The `send` operation returns once to the agent and once again to each clone. If the operation throws an exception other than Trip Exception, no clones have been constructed and only the agent experiences the failure. Otherwise the agent experiences the operation's success, and each clone separately and independently experiences the operation's success or failure.

A *clone* of an agent is a second agent that is a copy of the first, except that the identity, but not the authority, of the second differs from that of the first;

the permanent permits of the second may differ as well. For purposes of copying, the agent includes as properties the objects the agent owns.

Note. If several clones are sent to places that a particular engine sustains, the `send` operation transports a single encoding of the agent to that engine. Further savings (for example, of space) may be achieved at that engine.

Selecting a route

An agent can select, or help to select, the route it takes when it travels. The agent exerts such influence, as described in this section, using its ticket.

Ensuring a route

To travel with certainty between two places in the same region, an agent supplies on its ticket the destination's assigned telename and assigned teleaddress. If the destination is advertised, its teleaddress isn't required. In every case, the information above is sufficient to properly route the agent.

To travel with certainty between two places in different regions, an agent supplies routing advice by means of the assigned teleaddress. However, this information isn't sufficient in every case to properly route the agent.

As discussed earlier, a ticket may specify the way to a place. A *way* in turn may specify both a *means* of communication and the *authenticator* that the two engines that are engaged in communication use to identify one another.

Notes.

- Therefore the telesphere may have only an imperfect ability to route agents between places, especially places that are far removed from one another.
- Two subclasses of Means are defined next. Other subclasses are user-defined and perhaps region-specific, as are subclasses of Authenticator. A means may call for a particular wireless communication medium, for example.

Using reservable means

A *reservable means* is a means that can, but needn't, be reserved. A *reservation* is a statement of intent to use a means for one or more trips. A reservation expedites any trips that are actually made using the means.

The `reserve` operation cancels an existing reservation. If the *reservation interval* in seconds supplied as an argument is positive, the operation also makes a new reservation. If the operation is requested during a prior reservation interval, the new interval begins at once. Otherwise the new interval begins when the means is first actually used for a trip. The means is *reserved* before or during, but not after, the reservation interval.

A reservable means is merely a token for a reservation whose scope is the engine that constructs the means. Moving the means to another engine leaves the reservation inaccessible, and makes any indication of a reservation that the means might give subsequently unreliable.

Using existing connection means

An *existing connection means* is a reservable means that involves a previously established connection between two engines. However, the connection may no longer exist when a trip is actually taken using the means because neither of the two engines is obligated to maintain the connection on an ongoing basis. The connection is denoted using a *connection identifier*

An existing connection means is merely a token for a connection whose scope is the engine that constructs the means. Moving the means to another engine leaves the connection identifier undefined and the connection inaccessible.

Notes.

- By reserving the means, it is possible to increase the likelihood that the connection underlying an existing connection means will continue to exist.
- An engine can provide operations that manipulate the connection that a connection identifier denotes, but no such operation is part of the language.

Meeting other agents

One agent, the *petitioner*, can try to *meet* another agent, the *petitionee*, if the two occupy the same place. The *meeting* that results if the attempt succeeds gives each agent a reference to the other; thus the two agents can interact. At any time, either agent can *part* from the other, thereby ending their meeting.

An agent that meets with other agents is typically a member of Meeting Agent. Other agents behave as though they *were* meeting agents, but ones whose method for the `meeting` operation simply throws Meeting Denied.

Constructing a petition

A *petition* defines a meeting that an agent might ask to be arranged. The petition's main purpose is to identify the petitionee.

A petition's attributes provide any or all of the information in the following table. The first column identifies the attributes; the second describes them.

Identifier	*Attribute*
`agentClass`	The interface member class required of the petitionee.
`agentName`	The telename required of the petitionee.
`maximumWait`	The required maximum duration of the meeting arrangement, after which the petition *expires* (The same effect can be achieved using a temporary permit.)

Note. The petitionee is the one agent with a given name; any agent of a given authority; or any agent of a given class. These constraints can be combined.

Satisfying a petition

A petition constrains the petitionee, but not necessarily to a single agent. A petition is satisfied in principle by any agent with the name and class that the petition specifies—provided the agent occupies the place that the petitioner occupies. The petition is *satisfied* in practice by the one agent (if any) that agrees to meet with the petitioner before the petition expires.

The language requires the engine to wait for a petition to expire before ending its search for agents that satisfy it in principle. Thus an agent that enters after the petition is presented but before it expires may qualify.

The engine asks agents that satisfy the petition in principle to perform the `meeting` operation (see "Beginning a meeting"). The engine approaches these agents one after another, rather than in parallel; approaches them in an undefined order; and approaches each agent at most once.

Managing meetings

An agent is *acquainted* with the agents it meets. This section describes how acquaintances come and go and how the agent keeps track of them.

Beginning a meeting

An agent, using the `meet` operation, can meet the agent a petition specifies.

The engine mediates the beginning of a meeting. The engine requests the `meeting` operation of the petitionee. The operation's arguments identify the petitioner. If the operation succeeds, the meeting occurs. Otherwise it doesn't

The engine doesn't serialize the beginnings of meetings with an agent. Even if the agent is already performing the `meeting` operation, the engine isn't deterred from requesting the operation again on behalf of other petitioners.

Note The agent can provide its own serialization (for example, using a resource).

Ending a meeting

An agent, using the `part` operation, can concurrently part from one or more acquaintances or, using the `partAll` operation, can part from all of them.

The engine mediates the ending of a meeting. The engine sends a signal, a member of Part Event, whose source and intended destinations are the acquaintances. Each acquaintance receives the signal when and if it chooses.

Note If both the petitioner and the petitionee ask to part from one another, the earlier of the two requests ends the meeting, and thus the later request has no effect. Therefore only the agent whose request has no effect is sent a signal.

Keeping track of acquaintances

An agent designates an object as its *record* of a particular acquaintance. The petitioner supplies the object to the engine as an argument of the `meet` operation, while the petitionee supplies the object to the engine as the result of the `meeting` operation. The engine supplies the object to the petitioner or the petitionee—whichever is the case—as an attribute of the part event.

Note From time to time an agent may modify its record of an acquaintance so that the object reflects the state of the agent's interaction with that acquaintance.

Processes

The telesphere would be devoid of life if it weren't filled with agents and places. Both agents and places are *processes*. This chapter discusses processes, including how they identify and interact with one another.

Defining a process

A *process* is an object that can act autonomously and persistently. If the engine fails and later recovers, each process and the objects it owns have been preserved; the only effect upon them is their temporary unavailability.

Branding a process

A process has a *brand*, a mark a region applies to one or more processes. A region applies a new brand to each agent that enters the region from outside and an existing brand to each process constructed in the region—the brand of the process requesting its construction. Thus a brand denotes an agent that entered the region and any processes constructed in the region as a result.

The engine controls access to brands. A process's brand is its `regionalData` attribute. If the current sponsor is of the region's authority, an object can get or set the attribute of any process. Otherwise an object can get but not set it.

Note: Brands make possible a variety of process-tracking mechanisms but no such mechanism is provided by either the language or its predefined classes.

Phasing a process

A process has a lifetime which is divided into the following three phases:

1. The process performs the constructor at the request of its class. The class makes the request during its own performance of the `new` operation.

2. The process performs the `live` operation at the request of the engine. The argument is nil, which indicates that the process is being started.

3. The process is terminated by the engine.

The first phase can be interrupted. If the constructor throws an exception, the `new` operation fails. The process wasn't successfully constructed.

The second phase can be interrupted if the first phase isn't. If the `live` operation throws an exception, the engine terminates the current process (see "Terminating a process") unless the intersection of its permanent permits entitles it to be restarted, in which case the engine requests the `live` operation of it again, this time with the exception as argument.

Activating a process

A process has one or more *activations*, that is, threads of execution. When the engine asks a process to perform the `live` operation, the engine activates the process for that purpose. Thus the process passes through its second and third phases (but not its first) concurrently with other processes.

A process can have any number of concurrent activations and thus any number of execution threads. The engine activates a process to perform *any* sponsored operation that the engine requests of the process (for example, the

`entering` or `meeting`, as well as the `live`, operation). The activation ceases when the performance ends, whether successfully or unsuccessfully.

Note: A process isn't activated when an *object* asks it to perform a sponsored operation. The engine itself must make the request.

Prioritizing a process

A process has a *priority* which influences how quickly the process is executed. A process's priority is the minimum of its `desiredPriority` attribute and the `priority` attribute of the permit current when the process is current.

The engine preemptively schedules all processes except those that are blocked (for example, performing the `wait` operation). Thus the engine divides its computational resources among the various processes. Only if two processes differ in priority does the engine favor one over another.

The *normal priority* is 0. The priority of a process that the engine schedules only if it can schedule no higher-priority process is –20.

Note: Some regions let an OAM process exceed in priority any user process.

Terminating a process

The engine *terminates* the current sponsor, *S*, if the current object exhausts the current permit. The engine takes the following steps for *every* process activation that entails a method performed by *S* or an object it owns:

1. Redefine the current permit, as described elsewhere (see "Determining the current permit"), to rely upon the owner of the current object.

2. Throw Permit Exhausted. (This resumes performance of whatever method catches the exception. If *S* owns the object that performs that method, the engine simply throws the exception again.)

3. Return to Step 1 unless the activation no longer meets the above criteria.

The engine isolates and destroys *S* once it's not active. If the OAM policy in force requires, *S* is presented for diagnostic analysis before it's destroyed.

Whenever it terminates and doesn't restart a place, the engine terminates the occupants of that place as well as the place itself.

Note: Termination safeguards each object involved with *S* but not owned by it. Throwing an exception at each of the object's methods gives the object the opportunity to reestablish its invariant. Redefining the current permit in terms of the object's owner gives the object the resources it needs to do this.

Naming a process

A process has an *assigned telename* to distinguish it from other processes. More generally, a *telename* purports to denote either a single process (if the telename is assigned) or a set of *peers*, that is, processes of one authority.

A process's *authority* is the person or organization responsible for its actions. An engine place's authority is that of the region that contains it. The authority of any other process is that of a region, but not necessarily the one that contains the process: a region may *host* processes of other authorities.

Note: In other contexts, telenames denote packages, rather than processes.

Specifying an authority

A telename specifies an authority with an octet string. Chosen using the same algorithm throughout the telesphere, the octet string distinguishes the authority from other authorities—past, present, or future.

Note: Telescript technology recommends a particular algorithm for this purpose. Although beyond this manual's scope, the recommended algorithm mathematically computes the octet string, which is thus meaningless to a human being.

Specifying a process

A telename specifies a process with an octet string. Chosen using the same algorithm throughout the telesphere, the octet string distinguishes the process from other processes of the same authority.

Note: Telescript technology recommends a particular algorithm for this purpose. Although beyond this manual's scope, the recommended algorithm mathematically computes the octet string, which is thus meaningless to a human being.

Contacting a process through an operation

Processes can come into contact through operations. The passage of each argument or result of such an operation lets one process convey to the other either a reference to, a copy of, or the ownership of an object.

Note: This direct interaction is possible only if one process has a reference to the other. This isn't necessarily the situation. However, processes can also interact by means of packages, events, and resources.

Contacting a process through a package

Processes can come into contact through packages. A *package* is a dictionary that is protected and thus locked, and whose keys and values are locked as well. Like a process, a package has an assigned telename.

Note: Packages represent a means of multicast communication among processes within the scope of the same engine place.

Offering packages

A process can offer a package, either privately or publicly. A package that a process offers privately is available to that process alone. A package that a process offers publicly is available to that process but to other processes also.

The packages that a process *offers privately* form a list, the list that would result from appending to the process's `privatePackages` attribute the list of packages that the process offers publicly.

The packages that a process *offers publicly* form a second list, the list that would result from appending to the process's `publicPackages` attribute the list of packages offered publicly by the place the process occupies.

The packages that a process offers publicly or privately form an *effective package* whose keys reflect the union of the packages' keys. A key (and the associated value) in one package is included in the effective package in preference to a matching key (and value) in a package listed after it.

Searching packages for objects

The following operations search offered packages for objects in general:

- *Encoding or decoding*. The `encode` operation seeks objects that match objects that the operation might omit. The `decode` operation seeks objects whose names equal those of objects that are omitted. Both operations search packages that the current owner offers publicly.

- *Freezing or thawing*. The `freeze` operation seeks objects that match objects that the operation might freeze. The `thaw` operation seeks objects whose names equal those of objects that are frozen. Both operations search packages that the responder offers privately.

- *Upon request*. The `get`, `getPackage`, `find`, or `findPackage` operation seeks a specified object. All four operations search packages that the responder offers to the requester's owner.

All the operations discussed above search only the offered packages that two arguments select. The first, or `packages`, argument is either a set of telenames or nil. A set of telenames designates only the packages that the telenames denote. Nil designates all of the packages. The second, or `isNotInPackages`, argument is either a boolean or nil. *False* or nil selects the designated packages. *True* selects the undesignated packages.

The operations that search for objects upon request limit the two arguments. The `get` and `find` operations limit the `packages` argument to either a single telename or nil, and don't let the `isNotInPackages` argument be supplied, regarding it as nil instead. The `getPackage` and `findPackage` operations let neither argument be supplied, regarding both as nils.

Note: Other operations search offered packages for classes in particular.

Contacting a process through an event

Processes can come into contact through events. An *event* is an incident or condition as a particular process reports it by means of a signal.

Note: Events represent a means of multicast communication among processes within the scope of the same engine place.

Categorizing an event

Events fall into categories. Each category is denoted by Event or a subclass of Event. An event that falls into the category that a certain class denotes also falls into the categories that its interface superclasses denote.

An *event selector* selects zero or more events from among all events possible. A member of Event, an event selector selects events in the category that its class denotes. The event selector may include a telename. In this case, the event selector excludes signals sent by processes other than those named.

A *signal* is an event selector that reports an event by identifying a category into which the event falls; the process that sent the signal, identified by its assigned telename; and the time that the process sent the signal.

Sending a signal

One process *sends* a signal to zero or more processes. The one process is the *source* of the signal; the zero or more processes are its *intended destinations*

A signal is sent using the `signalEvent` operation. The current sponsor is considered the source of the signal. The signal's intended destinations are defined, relative to the operation's responder, by the signal's scope.

The following table shows the possible scopes of a signal. The first column identifies the scopes; the second defines them.

Identifier	*Definition*
`responder`	The responder.
`responderDeep`	The responder. If the responder is a place, its occupants and their occupants, recursively, as well.
`occupants`	If the responder is a place, its occupants. Otherwise no processes at all.
`occupantsDeep`	If the responder is a place, its occupants and their occupants, recursively. Otherwise no processes at all.

Enabling or disabling a signal

A process *enables* a signal it wishes to receive. Any one of a signal's intended destinations is an *actual destination* of that signal if that process enabled the signal before it was sent and didn't *disable* it later.

A process records the signals it currently enables as a set of event selectors. Two items in the set match if they are of the same class and restrict a signal's source in the same way. At any time the process can include an event selector in the set using the `enableEvents` operation, or exclude an event selector from the set using the `disableEvents` operation.

Receiving a signal

Whenever a process is sent a signal that the process currently enables, the engine includes that signal in a first-in first-out queue, which the engine maintains for the process. The signal remains in that queue indefinitely.

At a time it chooses, a process *receives* such a signal using the `getEvent` operation, thereby excluding the event from the first-in first-out queue of the process. The process presents an event selector and receives the selected signal that has been in the queue the longest. Alternatively, the process can empty the queue using the `clearEvents` operation.

Note: Thus the engine acts as an intermediary between a signal's source and its actual destinations.

Contacting a process through a resource

Processes can come into contact through resources. Although the engine performs atomically every predefined method except those for the `meet` and `wait` operations (either of which can block the current process), the engine doesn't necessarily perform user-defined methods with that assurance. A user-defined method can compensate for its lack of atomicity using a *resource*

Notes.

- Resources represent a means of multicast communication among processes within the scope of the same engine place.
- The engine performs even the `go` and `send` operations atomically.

Using a resource

A process (more precisely, a process activation) *uses* a resource to prevent other processes from using it in conflicting ways. To use a resource a process must present an unprotected reference to it. A process that requests use of a resource may be blocked until no other past or present use conflicts with its intended use. When several processes await conflicting use of the same resource, the engine grants use of the resource on a first-in first-out basis.

Note. Resources make possible the definition of critical conditional regions.

Using a resource exclusively

A process can make either shared or exclusive use of a resource. The *shared use* of a resource by one process conflicts with its concurrent exclusive use by another process, but not with its concurrent shared use. The *exclusive use* of a resource by one process conflicts with its concurrent use by another process, whether the latter use is shared or exclusive.

If a process asks for a resource's use while using it already, the second use is denied unless both uses are shared. If the second use is granted, the first use continues when the second use is completed, recursively.

Using a resource conditionally

At any moment in time a resource has a *condition*, which is among one or more possible conditions, each denoted by an identifier, that are defined when the resource is constructed. A process can examine a resource's condition at any time, but can set it only while using the resource exclusively.

A process can use a resource either conditionally or unconditionally. The *conditional use* of a resource by a process conflicts with a past or present use that left the resource in a condition other than one or more specified conditions. The *unconditional use* of a resource conflicts with a past or present use only under circumstances that are unrelated to condition.

Losing contact with a process

Two processes *lose contact* with one another when the engine voids all references in the closure of one to objects owned by the other.

Two processes lose contact with one another under various circumstances. When with the `go` operation an agent exits the place it occupies, it loses contact with all other processes. When with the `send` operation an agent's clone exits the place the agent occupies, it loses contact with all other processes. When with the `part` or `partAll` operation an agent parts from one or more acquaintances, it loses contact with those acquaintances. When a process is destroyed, it loses contact with all other processes.

Permits

Processes have considerable power in principle. In practice, that power is limited by means of permits. Programmers and administrators use permits to grant only certain capabilities to certain processes on certain occasions. This chapter discusses the capabilities that permits control.

Note: The principal purpose of permits is to prevent processes from consuming computer and communication resources in unintended amounts. This benefits both users, who construct the processes and perhaps pay for them, as well as providers, who provide the resources that the processes consume.

Defining a permit

The capabilities represented by the bulk of the language (for example, the arithmetic operations) are granted to every process implicitly. However, some capabilities are granted explicitly to certain processes but not to others.

A *permit* grants certain capabilities to a process, the permit's *subject*. These capabilities involve the subject's actions, resources, and forms of recognition.

Granting an action

Some capabilities grant actions to the subject. Such a capability is expressed as a boolean: *true* allows the action, while *false* disallows it.

A permit's attributes grant any or all of the actions in the following table. The first column identifies the attributes; the second describes the actions they control.

Identifier	*Action*
`canGo`	Go to another place.
`canSend`	Construct clones and send each to another place.
`canCreate`	Construct a peer process.
`canRestart`	Be restarted upon termination.
`canCharge`	Charge another process teleclicks (see "Granting a resource").
`canGrant`	Grant a capability to any of certain other processes.
`canDeny`	Deny a capability to any of certain other processes.

If it exhausts the current permit, the current process isn't restarted.

Note: The OAM policy in force may prevent the engine from restarting a process that it would otherwise have restarted.

Granting a resource

Some capabilities grant resources to the subject. Such a capability is expressed either as an integer or as nil. An integer grants a resource in the specified amount. Nil grants a resource in unlimited amount.

A permit's attributes grant any or all of the resources in the following table. The first column identifies the attributes; the second describes the resources they control.

Identifier	*Resource*
`age`	The number of seconds since the subject was constructed.
`charges`	The permit's *allowance*—that is, the number of *teleclicks* that are charged to the subject after its construction.
`extent`	The number of octets that is momentarily the subject's size.

Notes.

- The first of the three listed resources is elapsed time. Thus a permit can impose upon its subject a maximum lifetime.
- The expense in teleclicks of a particular service, whether rendered to one process by another or by the engine itself, can vary from place to place and time to time. The charges for some services (for example, space) can be assessed per unit of time.
- The resources described in the table aren't to be confused with members of the Resource class.

Granting a form of recognition

Some capabilities grant forms of recognition to the subject. Such a capability is expressed either as an integer or as nil. An integer grants recognition at the specified level. Nil grants recognition at the maximum level.

A permit's attributes grant any or all of the forms of recognition in the following table. The first column identifies the attributes; the second describes the forms of recognition they control; and the third defines, as mathematical intervals, the levels of recognition they admit.

Identifier	*Form of recognition*	*Levels*
`authenticity`	The subject's authenticity.	[0, 40]
`priority`	The subject's priority.	[–20, 20]

Receiving a permit

Several permits impinge on a process. Some of these permits are relatively permanent; others are more temporary. The *permanent permits* of a process are its native, regional, and local permits. The temporary permits of a process are discussed in "Receiving a temporary permit" later in this section.

Receiving a native permit

The *native permit* of a process, recorded by its `nativePermit` attribute, grants capabilities to the process as long as the process exists.

The native permit is set when the process is constructed. The current permit intervenes: the native permit must be equal to or before the current permit, and the native permit's allowance must be equal to or before what remains of the current permit's allowance. Provided it is finite, the native permit's allowance is charged to the current sponsor.

The engine limits access to native permits. An object can get or set the native permit of the current sponsor or any of its peers. However, only if the current permit grants the capability to grant or deny capabilities can setting the permit increase or decrease a capability, respectively. Furthermore, only the permit's allowance can be increased, and only by an amount that is equal to or before what remains of the current permit's allowance. If finite, the amount of the increase is charged to the current sponsor.

Note: Thus teleclicks are transferred between processes.

Receiving a regional permit

The *regional permit* of a process, recorded by its `regionalPermit` attribute, grants capabilities to the process as long as the process is in a region.

The regional permit is set when the process enters the region. If the `new` operation causes the process to enter, its regional permit is set to that of the current sponsor. If the `go` or `send` operation does so, the region sets the permit. The regional permit of an engine place is set to a basic permit.

The engine limits access to regional permits. If the current sponsor has the region's authority, an object can get or set the regional permit of any process. However, only if the current permit grants the capability to grant or deny capabilities can setting the permit increase or decrease a capability, respectively. If the current sponsor has another authority, an object can get only the current sponsor's regional permit and can set no regional permits.

Note: The language leaves undefined how the region initially sets the regional permit of a process that enters the region as a result of the `go` or `send` operation.

Receiving a local permit

The *local permit* of a process, recorded by its `localPermit` attribute, grants capabilities to the process as long as the process occupies a place.

The local permit is set when the process enters the place. If the `new` operation causes the process to enter, its local permit is set to that of the current sponsor. If the `go` or `send` operation does so, the place sets the permit. The local permit of an engine place is set to a basic permit.

The engine limits access to local permits. An object can get the local permit of the current sponsor or any occupant of the current sponsor. An object can set the local permit of any occupant of the current sponsor. However, only if the current permit grants the capability to grant or deny capabilities can setting the permit increase or decrease a capability, respectively.

Note: A place initially uses the `entering` operation to set the local permit of an occupant.

Receiving a temporary permit

A *temporary permit* of a process activation, recorded by none of the process's attributes, grants capabilities to the activation throughout its execution of a specified sequence of instructions. Temporary permits can be imposed in a nested fashion. Thus any number of such permits can be in force.

Note: Using a temporary permit, a process can hold in reserve a portion of its allowed amounts of resources. If it exhausts the bulk of any of those resources, the process can use the portion it held in reserve to take emergency action.

Reconciling permits

Sometimes the several permits that impinge on a process must be reconciled or altered. Reconciliation and alteration make use of the following concepts.

Intersecting two permits

Two permits can be intersected. The *intersection* is itself a permit, each of whose capabilities, C_0, is derived from the corresponding capabilities, C_1 and C_2, of the permits. Each capability is expressed as a boolean, integer, or nil.

C_0 is derived from C_1 and C_2 by the following steps:

1. If C_2 is nil, C_0 is C_1.

2. If C_1 is nil, C_0 is C_2.

3. C_0 is the minimum of C_1 and C_2.

Ordering two permits

Two permits can be ordered (see “Ordered”) by the following steps:

1. One permit is equal to the other if each capability of the first is equal to the corresponding capability of the second.

2. One permit is before the other if each capability of the first is either equal to or before the corresponding capability of the second.

3. One permit is after the other if each capability of the first is either equal to or after the corresponding capability of the second.

4. The two permits are unordered.

Ordering two capabilities

Two capabilities are ordered as follows (see “Ordered”). In practice, two ordered capabilities concern the same action, resource, or form of recognition.

Two capabilities that concern an action are ordered as are the associated booleans. Any other two capabilities are ordered as are the associated nils or integers. However, if one capability is nil and the other an integer, the result is as though an integer one greater were substituted for the nil.

Note: Thus one capability is before, equal to, or after another. Under no circumstances are two capabilities unordered.

Increasing or decreasing a capability

A capability is *increased* if replaced by another capability that is before it, *decreased* if replaced by another capability that is after it.

Enforcing the current permit

The engine enforces the one permit that results from reconciling the several permits that affect a process. That permit is defined and enforced as follows.

Determining the current permit

The *current permit*—except for its age capability—is the intersection of the permanent permits of the current sponsor and the temporary permits in force

from the performance of the most *tightly* enclosing sponsored operation. However, during the current sponsor's termination, the current permit is the intersection of the permanent permits of the current object's owner.

The current permit's age capability is the age capability of the intersection of the permanent permits of all processes performing operations as part of the current process activation and the temporary permits in force from the performance of the most *tightly* enclosing sponsored operation.

Violating the current permit

The current object *violates* the current permit if it tries to do any of the following, in which case the engine throws Permit Violated:

- Take an action that the current permit forbids.
- Use a resource in an amount that is after the amount that the current permit allows, without also exhausting the permit.
- Use more than what remains of the current permit's allowance, without also exhausting the permit.

Note: Capabilities that concern forms of recognition are irrelevant.

Exhausting the current permit

The current object *exhausts* the current permit by trying to do either of the following, in which case the engine terminates the current sponsor:

- Use a resource in an amount after that allowed by the intersection of the permanent permits upon which the current permit is based.
- Use more than what remains of the allowance of that intersection.

Note: Capabilities that concern actions and forms of recognition are irrelevant.

Patterns

This chapter describes the means the language provides for lexically analyzing and modifying strings.

Defining and using a pattern

A pattern is defined and used as follows.

Defining a pattern

A *pattern* lexically analyzes any given string, the pattern's *subject*. A certain pattern embodies a certain lexical requirement. The pattern can determine whether the subject *satisfies* the requirement and thus *matches* the pattern. The requirement itself takes the form of a string, the *text* of the pattern.

This section defines a pattern's text as a series of tokens, each zero or more characters. The text is formed by simply concatenating the tokens.

In general, the text of a pattern is a `Match` (see "Requiring a match"). The text of the simplest pattern is one character other than a metacharacter. The pattern matches a string that consists of only that one character. Presented individually throughout this chapter, *metacharacters* have special meanings.

Note: The metacharacters are "`$`", "`%`", "`(`", "`)`", "`*`", "`+`", "`–`", "`.`", "`?`", "`[`", "`\`", "`]`", "`^`", "`|`", and the characters that can appear in source programs only in escape sequences.

Using a pattern

A pattern performs the following three operations. The first operation searches a string for one substring that matches the pattern, the second and third for any number in succession. A *substring* is a sublist of a string:

- The `find` operation finds the first matching substring.
- The `split` operation finds all matching substrings.
- The `substitute` operation finds all matching substrings and substitutes for each a string perhaps tailored to the substring.

The `split` and `substitute` operations adhere to the following rules:

- The longest substring that could match does.
- No two matching substrings overlap.
- Even if the length of one matching substring is 0, the search for the next begins at the next position.

Requiring a match

A pattern's text can express the following matching requirements each of which is satisfied by zero or more characters of the pattern's subject.

Requiring a match

The requirement for a match is encoded as follows:

```
Match ::= AnchoredMatch [“|” Match]
```

This requirement is satisfied by any character that satisfies the requirement that an `AnchoredMatch` encodes.

Requiring an anchored match

The requirement for an anchored match is encoded as follows:

```
AnchoredMatch ::= [“^”] SuccessiveMatches [“$”]
```

This requirement is satisfied by any substring that satisfies the requirement that `SuccessiveMatches` encodes. If “ ^” or “$” occurs, the substring must begin or end the subject, respectively.

Requiring successive matches

The requirement for successive matches is encoded as follows:

```
SuccessiveMatches ::= RepeatedMatch [SuccessiveMatches]
```

This requirement is satisfied by as many contiguous substrings as there are `RepeatedMatch`'s if each substring satisfies the requirement that the corresponding `RepeatedMatch` encodes.

Requiring a repeated match

The requirement for a repeated match is encoded as follows:

```
RepeatedMatch ::= SingleMatch [“*”|“+”|“?”]
```

This requirement is satisfied by as many contiguous substrings as the following table dictates if each substring satisfies the requirement that `SingleMatch` encodes.

Terminal	*Number of Substrings*
“*”	0 or more
“+”	1 or more
“?”	0 or 1
None	1

Requiring a single match

The requirement for a single match is encoded as follows:

```
SingleMatch ::= “(” Match “)”
              | CharacterWithAttribute
              | CharacterInList
              | CharacterNotInList
              | CharacterWithName
              | Character
```

This requirement is satisfied by any substring that satisfies the requirement that the occurrence of one of the six nonterminals identified above encodes.

Requiring a character

A pattern's text can express the following matching requirements each of which is satisfied by a single character of the pattern's subject.

Requiring a character with certain attributes

The requirement for a character with certain attributes is encoded as follows:

```
CharacterWithAttribute ::=  "%" Attribute

             Attribute ::=  "A"|"D"|"L"|"P"|"S"|"U"|"7"
                         |  "a"|"d"|"l"|"p"|"s"|"u"
```

This requirement is satisfied by any character satisfying the predicate that `Attribute` selects from the following table. Each predicate decides whether the one, or both of the two, identified attributes of a character are *true*.

Character	*Predicate*
"A"	`isAlphabetic`
"D"	`isDecimalDigit`
"L"	`isLower`
"P"	`isPunctuation`
"S"	`isSpace`
"U"	`isUpper`
"7"	`isASCII`
"a"	`isASCII` and `isAlphabetic`
"d"	`isASCII` and `isDecimalDigit`
"l"	`isASCII` and `isLower`
"p"	`isASCII` and `isPunctuation`
"s"	`isASCII` and `isSpace`
"u"	`isASCII` and `isUpper`

Requiring a character in a certain list

The requirement for a character in a certain list is encoded as follows:

```
CharacterInList ::=  "[" List "]"

           List ::=  Item [List]
           Item ::=  CharacterInInterval
                  |  CharacterWithName
```

This requirement is satisfied by any character that satisfies the requirement encoded by one of the `Item`'s in the `List`.

Requiring a character not in a certain list

The requirement for a character not in a certain list is encoded as follows:

```
CharacterNotInList ::= "[" "^" List "]"
```

This requirement is satisfied by any character that satisfies the requirement encoded by none of the `Item`'s in the `List`.

Requiring a character in a certain interval

The requirement for a character in a certain interval is encoded as follows:

```
CharacterInInterval ::= CharacterWithName1 "–" CharacterWithName2
```

This requirement is satisfied by any character that is neither before the character that `CharacterWithName`$_1$ requires nor after the character that `CharacterWithName`$_2$ requires.

Requiring a character with a certain name

The requirement for a character with a certain name is encoded as follows:

```
CharacterWithName ::= "\" METACHARACTER | NONMETACHARACTER
```

This requirement is satisfied by either the metacharacter that METACHARACTER encodes or the non-metacharacter that NONMETACHARACTER encodes.

Notes.

- A metacharacter immediately preceded by a "\" avoids its special meaning.
- When a pattern's text occurs in a source program (in the denotation of a string), each "\" in the text must be doubled. Thus "\\.", not "\.", denotes a character.

Requiring a character

The requirement for a character is encoded as follows:

```
Character ::= "."
```

This requirement is satisfied by any character.

Calendar times

This chapter describes the means the language provides for lexically analyzing and modifying times.

Defining a calendar time

A *calendar time* identifies a date and time of day to the precision of 1 second. It uses Coordinated Universal Time (*UTC*) for this purpose. A calendar time also records a local time zone's *permanent offset* in minutes from UTC and its *seasonal offset* in minutes from its permanent offset. A seasonal offset other than 0 signifies Daylight Savings Time (*DST*).

Notes.

- UTC is effectively what used to be known as Greenwich Mean Time (GMT).
- Because some engines internally represent times more compactly than calendar times, times are better for storing and transporting dates and times.

Accessing a calendar time

A calendar time exposes to individual examination and modification the year in the Gregorian calendar, the month of that year, the day of that month, the hour of that day, the minute of that hour, the second of that minute, the permanent and seasonal offsets, and the days of the week and year.

Accessing the time

A calendar time's attributes expose to examination and modification the facets of the time in the following table. Each attribute is either an integer or nil. The first column of the table identifies the attributes; the second describes the facets they represent; and the third gives the mathematical intervals into which the attributes normally fall when they are integers.

Identifier	*Facet*	*Interval*
`hour`	The hour of the day.	[0, 23]
`minute`	The minute of the hour.	[0, 59]
`second`	The second of the minute.	[0, 60] [1]
`zone`	The permanent offset.	[–720, 720]
`dst`	The seasonal offset.	[–720, 720]

[1] Second 60 accounts for leap seconds.

Accessing the date

A calendar time's attributes expose to examination and modification the facets of the date in the following table. Each attribute is either an integer or nil. The first column identifies the attributes; the second describes the facets they represent; and the third gives the mathematical intervals into which the attributes normally fall when they are integers.

Identifier	*Facet*	*Interval*
`year`	The year in the Gregorian calendar (for example, 1995).	[∞,∞]
`month`	The month of the year.	[1, 12]
`day`	The day of the month.	[1, 31]
`dayOfWeek`	The day of the week. Sunday is 1, Monday is 2, Tuesday is 3, Wednesday is 4, Thursday is 5, Friday is 6, and Saturday is 7.	[1, 7]
`dayOfYear`	The day of the year.	[1, 366] [1]

[1] Day 366 accounts for leap years.

Normalizing a calendar time

A facet of a calendar time can lie outside its normal interval. For example, a calendar time might identify the date as September 32, meaning October 2, as the result of adding 2 to the day of the month. Such abnormalities are useful but must be eliminated before the calendar time can be considered correct. A calendar time is made correct in this sense by normalization.

A calendar time is *normalized* by normalizing the time and then the date.

Normalizing the time

The time is *normalized* by the following steps:

1. If nil, the hour, minute, or second is made 0. If nil, either offset is made that of a normalized calendar time that denotes the current time.

2. The permanent or seasonal offset is replaced by the result of transposing into the closed interval [–720, 720] the remainder left by dividing the offset by 24*60.

Note: Even if, for example, the unnormalized and normalized dates disagree as to whether DST is in effect, the seasonal offset isn't further adjusted.

Normalizing the date

The date is *normalized* by the following steps:

1. If nil, the year, month, or day is made 1.

2. The second is placed in its normal interval by doing one of two things repeatedly: adding 60 to the second and subtracting 1 from the minute, or subtracting 60 from the second and adding 1 to the minute.

3. The minute is placed in its normal interval by doing one of two things repeatedly: adding 60 to the minute and subtracting 1 from the hour, or subtracting 60 from the minute and adding 1 to the hour.

4. The hour is placed in its normal interval by doing one of two things repeatedly: adding 24 to the hour and subtracting 1 from the day, or subtracting 24 from the hour and adding 1 to the day.

5. The month is placed in its normal interval by repeatedly either adding 12 to the month and subtracting 1 from the year, or subtracting 12 from the month and adding 1 to the year. Then the day is placed in the interval [1, *K*] by either adding *K* to the day and subtracting 1 from the month, or subtracting *K* from the day and adding 1 to the month. *K* is the number of days in the subject month in the subject year. If one addition or subtraction of *K* doesn't place the day in [1, *K*], this entire step is repeated.

6. The days of the week and year are set correctly.

Part Four— Predefined Classes

This part of the manual defines the language's predefined classes. The classes form the following module. The ellipsis in the module's definition stands for the interface definitions that appear in the following chapters:

```
Telescript: module = ( ... );
```

The predefined classes are positioned in the class graph as follows:

Object
• Authenticator
• Calendar Time
• Class (Protected)
• Class Name
• Collection (Compared)
• • List (Ordered)
• • • Bit String
• • • Octet String
• • • Stack
• • • String (Cased)
• • Set (Verified)
• • • Dictionary
• • • • Package (Named and Protected)
• Event
• • Process Event
• • • Exit Event
• • • • Death Event
• • • Part Event
• Exception
• • Meeting Exception
• • Programming Exception
• • • Class Exception
• • • Collection Exception
• • • Kernel Exception
• • • • Execution Exception
• • • Miscellaneous Exception
• • • Primitive Exception
• • • Process Exception
• • Trip Exception
• Iterator
• Means
• • Existing Connection Means (Reservable Means)
• Pattern
• Permit
• Petition
• Primitive (Protected)
• • Bit (Ordered)
• • Boolean (Ordered)
• • Character (Cased and Ordered)
• • Identifier (Ordered)
• • Nil
• • Number (Ordered)
• • • Integer
• • • Real
• • Octet (Ordered)
• Process (Named, Package Process

Cased
Compared
• Equal
• Same
Event Process
Meeting Agent
Meeting Place
Named
Ordered
Package Process
Permit Process
Protected
Reservable Means
Uncopied
Unmoved
Verified

Legend

The classes on the left above are the predefined flavors. A flavor's immediate interface superclasses are indicated as follows. If the superclass is itself a flavor, it appears above the class and indented one degree less. If the superclass is a mix-in, it appears beside the flavor enclosed in parentheses.

The classes on the right are the predefined mix-ins. A mix-in's immediate interface superclasses appear above it and indented one degree less.

The predefined classes are defined individually and in alphabetical order in the chapters that follow. Each chapter, after relocating its class in the class graph, includes as needed some or all of the sections in the following table.

Section	*Purpose*
Class	Defines the class, apart from its features, and how the class implements and thus specializes inherited features.
Subclasses	Defines the predefined immediate subclasses that don't have their own sections of the manual.
Constructor	Explicitly redefines the constructor for class instances.
Attributes	Defines the native attributes of the class.
Operations	Defines the native operations of the class except for conversions.
Conversions	Defines the instance operations that perform conversions.
Sealings	Identifies any inherited features that the class seals.

Agent

Object
- Process (Named, Package Process, Permit Process, and Uncopied)
  - **Agent**

Class

```
Agent: abstract interface (Process) =
   (
      private
         see go, send
   );
```

A process that can move from place to place.

Specialization. The responder's `contacts` attribute consists of unprotected references to the responder's acquaintances. The engine includes a reference when a meeting begins, and excludes any such reference when the meeting ends.

Operations

```
go: sealed op (ticket: owned Ticket) TicketStub
   throws PermitViolated, ProcessNotControlled,
      ReferenceProtected, TripException;
```

Takes the responder on the trip the `ticket` argument defines. Returns the ticket stub that documents the trip.

Exceptions.

- Throws `PermitViolated` if the current permit forbids the `go` operation.
- Throws `ProcessNotControlled` unless the current process is the responder not only for the sponsored operation that it performs at the engine's request, but also for all operations entailed by that operation.
- Throws `ReferenceProtected` if the responder is protected.
- Throws `TripException` if the trip fails.

```
send: sealed op (
   tickets: owned List[Ticket, Equal];
   charges: Integer /* nonnegative */ |Nil) TicketStub|Nil
   throws PermitViolated, ProcessNotControlled,
      ReferenceProtected, TripException;
```

Takes clones of the responder on the trips the `tickets` argument defines. Returns nil to the responder and a ticket stub to each clone. A clone's native permit equals the current permit except for its `charges` attribute; the operation sets that attribute to the `charges` argument.

Exceptions.

- Throws `PermitViolated` if the current permit forbids the `send` operation or its `charges` attribute is insufficient.
- Throws `ProcessNotControlled` unless the current process is the responder not only for the sponsored operation that it performs at the engine's request, but also for all operations entailed by that operation.
- Throws `ReferenceProtected` if the responder is protected.
- Throws `TripException` if or the trip fails.

Authenticator

Object
- **Authenticator**

Class

```
Authenticator: interface =
   (
      public
         see initialize, securityRegime
   );
```

An object that determines how the telesphere authenticates an agent—for example, as the agent passes between regions in the course of a trip.

Constructor

```
initialize: op (securityRegime: Telename /* assigned */ );
```

Sets the responder's attributes according to the following table.

Attribute	*Argument*	*Default*
securityRegime	Yes	–

Attributes

```
securityRegime: Telename /* assigned */ ;
```

The security regime that the responder dictates.

Bit

Object
• Primitive (Protected)
• • **Bit** (Ordered)

Class

```
Bit: sealed interface (Primitive, Ordered) = ();
```

A lightweight primitive with two possible values, *zero* and *one*

Specializations.

- The `copy` operation regards the responder's value as its only property.
- The `order` operation relates *zero* and *one* as it does integers 0 and 1.

Bit String

Object
• Collection (Compared)
• • List (Ordered)
• • • **Bit String**

Class

```
BitString: sealed interface (List[Bit, Equal]) =
   (
      public
         see asOctetString, initialize
   );
```

A list whose items are bits.

Specialization. The `order` operation relates two bit strings as though first enough *zero* bits were prepended to one bit string to make it equal in length to the other. A bit string is unrelated to an instance of any other class.

Constructor

```
initialize: op (
   bits: protected Object /* Bit|BitString */ ...);
```

Sets the responder's items to the bits the arguments contribute. The positions of the items reflect their order of contribution. A bit contributes itself. A bit string contributes its items in order of increasing position. The arguments contribute in order of increasing signature position.

Conversions

```
asOctetString: op () OctetString;
```

Returns an octet string whose length is i where the responder's length, n, is in the interval $[8i–7, 8i]$. Bit j of the octet at position k is determined as follows. If $(8k–j) \leq n$, the bit equals that at position $(8k–j)$. Otherwise the bit is zero.

Boolean

Object
• Primitive (Protected)
• • **Boolean** (Ordered)

Class

```
Boolean: sealed interface (Primitive, Ordered) =
  (
    public
      see and, not, or
  );
```

A lightweight primitive with two possible values, *false* and *true*.

Specializations.

- The copy operation regards the responder's value as its only property.
- The order operation relates *false* and *true* as it does integers 0 and 1.

Operations

```
and: op (boolean: Boolean) Boolean;
```

Returns the logical conjunction of the responder and the boolean argument.

```
not: op () Boolean;
```

Returns the logical negation of the responder.

```
or: op (boolean: Boolean) Boolean;
```

Returns the logical disjunction of the responder and the boolean argument.

Calendar Time

Object
- **Calendar Time**

Class

```
CalendarTime: interface =
  (
    public
      see asTime, day, dayOfWeek, dayOfYear, dst, hour, initialize,
        minute, month, normalize, second, year, zone
  );
```

An object that identifies a date and time of day to the precision of 1 second using UTC . A calendar time also identifies a time zone and to what extent, if any, DST is in effect in that zone at the indicated time.

Constructor

initialize: `op ();`

Sets the responder's attributes according to the following table.

Attribute	*Argument*	*Default*
`day`	–	nil
`dayOfWeek`	–	nil
`dayOfYear`	–	nil
`dst`	–	nil
`hour`	–	nil
`minute`	–	nil
`month`	–	nil
`second`	–	nil
`year`	–	nil
`zone`	–	nil

Attributes (date)

day: `Integer|Nil;`

The day of the month the responder identifies.

dayOfWeek: `Integer|Nil;`

The day of the week the responder identifies.

dayOfYear: `Integer|Nil;`

The day of the year the responder identifies.

---

month: `Integer|Nil;`

The month of the year the responder identifies.

---

year: `Integer|Nil;`

The year in the Gregorian calendar the responder identifies.

Attributes (time)

dst: `Integer|Nil;`

The seasonal offset in minutes the responder identifies.

---

hour: `Integer|Nil;`

The hour of the day the responder identifies.

---

minute: `Integer|Nil;`

The minute of the hour the responder identifies.

---

second: `Integer|Nil;`

The second of the minute the responder identifies.

---

zone: `Integer|Nil;`

The permanent offset in minutes the responder identifies.

Operations

```
normalize: op () Boolean
   throws ReferenceProtected;
```

Normalizes the responder. If the responder is thereby modified, returns *true*.

Exception Throws `ReferenceProtected` if the responder is protected.

Conversions

asTime: `op () Time;`

Returns the time the responder would identify if it were normalized.

Cased

Cased

Class

```
Cased: mixin interface =
   (
      public
         see isLower, isUpper, makeLower, makeUpper
   );
```

An object that incorporates characters and that lets the distinction between uppercase and lowercase characters be observed.

Attributes

```
isLower: abstract readonly Boolean;
```

True if the responder includes any lowercase characters.

```
isUpper: abstract readonly Boolean;
```

True if the responder includes any uppercase characters.

Operations

```
makeLower: abstract op () Cased;
```

Returns a copy of the responder in which all uppercase characters have been replaced with their lowercase equivalents.

```
makeUpper: abstract op () Cased;
```

Returns a copy of the responder in which all lowercase characters have been replaced with their uppercase equivalents.

Character

Object
- Primitive (Protected)
- • **Character** (Cased and Ordered)

Class

```
Character: sealed interface (Primitive, Cased, Ordered) =
  (
     public
        see asInteger, isAlphabetic, isASCII, isDecimalDigit,
           isPunctuation, isSpace
  );
```

A lightweight `primitive whose possible values are the Unicode characters. The class has one instance for each of the 65,535 possible Unicode values.

Note. The instances of this class include two noncharacters.

Specializations.

- The `copy` operation regards the responder's value as its only property.
- The `isLower` and `isUpper` attributes and the `makeLower` and `makeUpper` operations use the responder's case.
- The `makeUpper` operation, if requested of "β", returns "S", not two of those characters as Unicode requires.
- The `order` operation relates two characters as it does the integers that are their Unicode values.

Attributes

isAlphabetic: `readonly Boolean;`

True if the responder is an alphabetic character.

isASCII: `readonly Boolean;`

True if the responder is an ASCII character.

isDecimalDigit: `readonly Boolean;`

True if the responder is a decimal digit character.

isPunctuation: `readonly Boolean;`

True if the responder is a punctuation character.

isSpace: `readonly Boolean;`

True if the responder is a space character.

Conversions

asInteger: `op () Integer` */* [0, 65535] */* `;`

Returns the responder's Unicode value.

Class

Object
- **Class** (Protected)

Class

```
Class: sealed interface (Object, Protected) =
  (
    public
      see initialize, isSubclass, isSubclassByName, name, new
  );
```

An object that defines other objects, the instances of the class.

Specialization. The copy operation, if the responder is defined, regards the responder's immediate interface and implementation superclasses as properties. If the responder is derived, the operation regards its class family as a property.

Constructor

```
initialize: op ()
  throws FeatureUnavailable;
```

Exception. Throws `FeatureUnavailable` if the feature is requested.

Attributes

```
name: readonly protected ClassName!;
```

The responder's class name.

Operations

```
isSubclass: op (_class: Class) Boolean;
```

Returns *true* if the responder is the `class` argument or an interface subclass thereof.

```
isSubclassByName: op (_class: protected ClassName) Boolean;
```

Returns *true* if the responder is the `class` argument's subject or an interface subclass thereof.

```
new: op (parameters: Object ...) Object
   throws ClassAbstract, Exception, ObjectUninitialized;
```

Returns the new instance of the responder defined by the `parameters` arguments, the arguments of the constructor as redefined by the responder.

If the responder is a class family, the operation behaves as though it were requested of the class derived with the default parameters.

Exceptions.

- Throws `ClassAbstract` if the responder is abstract.
- Throws `Exception` if the constructor fails.
- Throws `ObjectUninitialized` if the constructor fails.

Class Exception

Object
• Exception
• • Programming Exception
• • • **Class Exception**

Class

```
ClassException: abstract interface
   (ProgrammingException) = ();
```

A programming exception thrown during the construction of a class.

Subclasses

```
SuperclassesInvalid: interface (ClassException) = ();
```

The class definition designates superclasses invalidly or inconsistently.

Class Name

Object
- **Class Name**

Class

```
ClassName: sealed interface =
   (
      public
         see classDigest, initialize
   );
```

An object that denotes a class, its *subject*, distinguishing the class from all other classes, predefined or user-defined.

Constructor

```
initialize: op (classDigest: owned OctetString);
```

Sets the responder's attributes according to the following table.

Attribute	*Argument*	*Default*
classDigest	Yes	–

Attributes

```
classDigest: owned OctetString;
```

An octet string that denotes the responder's subject.

Note: Telescript technology recommends a particular algorithm for choosing this octet string. The recommended algorithm is beyond this manual's scope.

Collection

Object
• **Collection** (Compared)

Class

```
Collection: interface[Item: Class; Match: Class<:Compared]
    (Object, Match) =
(
   public
      see asList, clear, examine, exclude, include, initialize,
         iterator, length, shallowCopy
);
```

An object that includes zero or more objects, the collection's *items*. The class doesn't order the items, but a subclass can do so. The number of items is the collection's *length*. The length of an *empty* collection is 0.

This is a class family. The `Item` parameter is the required interface member class of each item of every member of a derived class. The `Match` parameter is the interface member class required to decide whether two items *match*.

Note. A collection's length is unbounded.

Specialization. The `copy` operation regards the responder's items as properties. Two collections are copy-equal if their items can be paired so that the two items in each pair are copy-equal.

Constructor

```
initialize: op (items: Item ...);
```

Sets the responder's items to the arguments using the `include` operation.

Attributes

```
length: readonly Integer /* nonnegative */ ;
```

The responder's length.

Operations (modification)

clear: `op ()`
`throws ReferenceProtected;`

Removes from the responder and discards all of the responder's items.

Exception Throws `ReferenceProtected` if the responder is protected.

---

exclude: `op (item: protected Object) Item|Nil`
`throws ReferenceProtected;`

Excludes from the responder and returns any of the items that match the `item` argument. If there are no such items, returns nil.

Exception Throws `ReferenceProtected` if the responder is protected.

---

include: `op (item: Item)`
`throws ReferenceProtected;`

Includes the `item` argument in the responder as a new item.

Exception Throws `ReferenceProtected` if the responder is protected.

Operations (examination)

examine: `op (item: protected Object) Item|Nil;`

Returns any item of the responder that matches the `item` argument. Leaves the responder unchanged. If there are no such items, returns nil.

---

iterator: `op () unprotected Iterator[Item];`

Returns a new iterator whose items are the responder's.

Modifying the responder or transporting it to another place renders the iterator internally inconsistent: subsequently, the iterator can fail to produce an item it would have produced, can produce an item a second time, or both.

Note The result's class, a subclass of Iterator, is undefined.

---

shallowCopy: `op () Collection[Item, Match];`

Returns a second instance of the responder's class that has the same items.

Conversions

asList: `op () List[Item, Match]!;`

Returns a list whose items are those of the responder.

Collection Exception

Object
• Exception
• • Programming Exception
• • • **Collection Exception**

Class

```
CollectionException: abstract interface
   (ProgrammingException) = ();
```

A programming exception thrown by a collection.

Subclasses

```
KeyInvalid: interface (CollectionException) = ();
```

A purported dictionary key isn't actually a key, or two proposed keys match.

```
PositionInvalid: interface (CollectionException) = ();
```

A purported list position isn't actually a position.

```
StackDepleted: interface (CollectionException) = ();
```

The length of a stack prevents its manipulation.

Compared

Compared

Class

```
Compared: mixin interface =
   (
      public class
         see compare
   );
```

An object that decides whether two objects *match*.

Specialization. The `compare` operation behaves as it does for Equal.

Operations

```
compare: op (object1, object2: protected Object)
      Identifier /* as enumerated */ ;
```

Returns the identifier that relates the `object1` argument to the `object2` argument. The identifier is one of those defined by Ordered. For example, if the first argument is before the second, the identifier is `before`.

Death Event

Object
• Event
• • Process Event
• • • Exit Event
• • • • **Death Event**

Class

```
DeathEvent: interface (ExitEvent) =
  (
    public
      see exception, initialize
  );
```

An exit event that involves the termination of a process.

Constructor

initialize: op (source: Telename|Nil; time: Time|Nil);

Sets the responder's attributes according to the following table.

Attribute	*Argument*	*Default*
source	Yes	nil
time	Yes	nil
exception	–	nil
record	–	nil

Attributes

exception: readonly Exception|Nil;

The exception that caused the process termination that the responder involves. If the process terminated voluntarily, nil.

Dictionary

Object
• Collection (Compared)
• • Set (Verified)
• • • **Dictionary**

Class

```
Dictionary: interface[Key, Value: Class; Match: Class<:Compared]
   (Set[Key, Match]) =
   (
      public
         see add, drop, find, get, initialize, rekey, set, transpose
   );
```

A set whose items are associated one-to-one with other objects. The items are the dictionary's *keys*; the other objects are its *values*. If its reference to a key is ever voided, a dictionary automatically excludes that key and its value.

This is a class family. The `Key` parameter is the required interface member class of each key of every member of a derived class. The `Value` parameter is the required interface member class of each value. The `Match` parameter is the interface member class required to decide whether two keys *match*. Equal decides whether two values *match*.

Specializations.

- The `clear` operation removes and discards the responder's keys and values.
- The `copy` operation regards the responder's keys and values as properties. Two dictionaries are copy-equal if their keys can be paired so that the two keys in each pair are copy-equal and if their values are copy-equal as well.
- The `difference`, `exclude`, or `intersection` operation excludes from the responder the value of each excluded key.
- The `include` operation uses the `set` operation to make the included key's value nil. If the responder rejects nils, the operation throws an exception.
- The `shallowCopy` operation uses neither the `new` nor the `include` operation to construct its result, which includes the same keys and values as the responder includes.
- The `union` operation includes in the responder each included key's value. If the argument isn't a dictionary, the operation throws Argument Invalid.

Constructor

```
initialize: op (keysAndValues: Object ...
   /* (even numbered) key: Key; value: Value */);
```

Sets the responder's keys and values to the arguments. The keys and values are added, using the `set` operation, in order of increasing signature position.

Operations (modification)

add: `op (key: Key; value: Value)`
`throws KeyInvalid, ReferenceProtected;`

Includes the `key` and `value` arguments in the responder as a new key and its value, respectively.

Exceptions.

- Throws `KeyInvalid` if the new key matches an existing key.
- Throws `ReferenceProtected` if the responder is protected.

---

drop: `op (key: protected Key) Value`
`throws KeyInvalid, ReferenceProtected;`

Excludes from the responder and discards the key that matches the `key` argument. Excludes and returns the value associated with the discarded key.

Exceptions.

- Throws `KeyInvalid` if the object doesn't match an existing key.
- Throws `ReferenceProtected` if the responder is protected.

---

rekey: `op (currentKey: protected Key; newKey: Key)`
`throws KeyInvalid, ReferenceProtected;`

Discards the responder's key that matches the `currentKey` argument and substitutes for that key the `newKey` argument.

Exceptions.

- Throws `KeyInvalid` if the `currentKey` argument doesn't match an existing key or the `newKey` argument does.
- Throws `ReferenceProtected` if the responder is protected.

---

set: `op (key: Key; value: Value)`
`throws ReferenceProtected;`

Includes the `key` and `value` arguments in the responder as a new key and its value, respectively. Before doing so, excludes and discards any existing key that matches the new key and discards the associated value.

Exception Throws `ReferenceProtected` if the responder is protected.

---

transpose: `op (key1, key2: protected Key) throws KeyInvalid, ReferenceProtected;`

Interchanges the keys that match the `key1` and `key2` arguments.

Exceptions.

- Throws `KeyInvalid` if either specified key doesn't match an existing key.
- Throws `ReferenceProtected` if the responder is protected.

Operations (examination)

find: `op (value: protected Value) Key|Nil;`

Returns any key of the responder whose value matches the `value` argument. Leaves the responder unchanged. If there is no such key, returns nil.

---

get: `op (key: protected Key) Value throws KeyInvalid;`

Returns the value of the responder whose key matches the `key` argument. Leaves the responder unchanged.

Exception Throws `KeyInvalid` if there is no such key.

Engine Place

Object
- Process (Named, Package Process, Permit Process, and Uncopied)
  - Place (Unmoved)
    - **Engine Place**

Class

```
EnginePlace: interface (Place) =
   (
      public
         see initialize
      private
         see dst, zone
   );
```

A place that represents an engine.

Constructor

```
initialize: op (
    name: copied Telename /* assigned */ ;
    address: copied Teleaddress /* assigned */ ;
    zone, dst: Integer /* [–720, 720] */ |Nil)
  throws AddressInvalid, FeatureUnavailable;
```

Sets the responder's attributes according to the following table.

Attribute	*Argument*	*Default*
`name`	Yes	–
`address`	Yes	–
`zone`	Yes	0
`dst`	Yes	0
`age`	–	0
`assignments`	–	0
`charges`	–	0
`contacts`	–	empty
`desiredPriority`	–	0
`isPossibleDuplicate`	–	*false*
`localPermit`	–	basic
`nativePermit`	–	basic
`privatePackages`	–	empty
`publicPackages`	–	empty
`regionalData`	–	nil
`regionalPermit`	–	basic

Note. The language doesn't expose the aspects of Place that let the constructor set the `name` and `address` attributes of a place arbitrarily.

Exceptions.

- Throws `AddressInvalid` if the `authority` attributes of the name and address aren't copy-equal.
- Throws `FeatureUnavailable` if an engine place exists already.

Attributes

```
dst: Integer /* [–720, 720] */ ;
```

The seasonal offset in minutes of the time at the responder.

```
zone: Integer /* [–720, 720] */ ;
```

The permanent offset in minutes of the time at the responder.

Equal

Compared
• **Equal**

Class

```
Equal: mixin interface (Compared) =
   (
      public class
         see compare
   );
```

A compared object that matches objects that are copy-equal.

Sealings

```
compare: sealed;
```

If the objects are copy-equal, return `equal`. Otherwise return `unordered`.

Event

Object
- **Event**

Class

```
Event: interface =
   (
      public
         see initialize, source, time
   );
```

An object that selects zero or more events or reports one of them.

Constructor

```
initialize: op (source: Telename|Nil; time: Time|Nil);
```

Sets the responder's attributes according to the following table.

Attribute	*Argument*	*Default*
source	Yes	nil
time	Yes	nil

Attributes

```
source: sealed Telename|Nil;
```

The source of the events the responder selects.

```
time: sealed Time|Nil;
```

The time of the events the responder selects.

Event Process

Event Process

Class

```
EventProcess: mixin interface (Process) =
   (
      public
         see signalEvent
      private
         see clearEvents, disableEvents, enableEvents, getEvent
   );
```

A process that can be sent signals and that can receive them.

Specialization. The constructor empties the responder's set of event selectors and the responder's first-in first-out queue of signals.

Operations (enabling and disabling)

```
disableEvents: sealed op (selector: protected Event|Nil)
   throws ReferenceProtected;
```

Excludes from the responder's set of event selectors any item that matches the `selector` argument. If nil is supplied, the set is emptied.

Exception. Throws `ReferenceProtected` if the responder is protected.

```
enableEvents: sealed op (selector: protected Event)
   throws ReferenceProtected;
```

Includes the event `selector` argument in the responder's set of event selectors unless the event selector matches an existing item of that set.

Exception. Throws `ReferenceProtected` if the responder is protected.

Operations (sending and receiving)

```
clearEvents: sealed op ()
   throws ReferenceProtected;
```

Empties the responder's first-in first-out queue of signals and makes the responder subsequently disregard any event sent before the current time.

Exception. Throws `ReferenceProtected` if the responder is protected.

```
getEvent: sealed op (
   maximumWait: Integer|Nil /* nonnegative */ ;
   selector: Event|Nil) Event|Nil
      throws ReferenceProtected;
```

Removes from the responder's first-in first-out queue and returns the signal first included in the queue. If the `selector` argument isn't nil, considers for removal only signals that report events that the event selector selects.

If the `maximumWait` argument isn't nil, the operation waits for a signal the specified number of seconds. If a signal remains unavailable, the result is nil. If the argument is nil, the result isn't: the wait is indefinite.

Exception Throws `ReferenceProtected` if the responder is protected.

```
signalEvent: sealed op (
   selector: protected Event;
   scope: Identifier /* as enumerated */ |Nil)
      throws ReferenceProtected;
```

Sends the signal formed by copying the `selector` argument, setting its `source` and `time` attributes authentically, and locking the modified copy. However, if the `time` attribute is a time in the future, the operation delays sending the signal until that time. The signal has the scope that the `scope` argument specifies. If the argument is nil, the scope is `responderDeep`.

Exception Throws `ReferenceProtected` if the responder is protected.

Exception

Object
- **Exception**

Class

```
Exception: abstract interface = ();
```

An object that describes the failure of an operation.

Execution Exception

Object
• Exception
• • Programming Exception
• • • Kernel Exception
• • • • **Execution Exception**

Class

```
ExecutionException: abstract interface
   (KernelException) = ();
```

A kernel exception that indicates that the engine can't execute an object.

Subclasses

```
ArgumentInvalid: interface (ExecutionException) = ();
```

An argument of an operation doesn't satisfy its type.

```
ArgumentMissing: interface (ExecutionException) = ();
```

An argument of an operation is missing.

```
AttributeReadOnly: interface (ExecutionException) = ();
```

An attribute is read only and thus can't be set.

```
ClassUnavailable: interface (ExecutionException) = ();
```

A class is unavailable.

```
EscalationInvalid: interface (ExecutionException) = ();
```

A feature is escalated improperly.

```
FeatureUnavailable: interface (ExecutionException) = ();
```

A feature is undefined or private and the responder isn't nil.

InternalException: `interface (ExecutionException) = ();`

The engine doesn't implement some aspect of the language.

Note: Ideally, the engine never throws a member of this class. Practically, the engine's specification states under what conditions the engine does so.

PermitExhausted: `interface (ExecutionException) = ();`

The current permit is exhausted.

PropertyUndefined: `interface (ExecutionException) = ();`

A property identifier is undefined.

ReferenceProtected: `interface (ExecutionException) = ();`

A reference is protected and thus the object can't be modified.

ReferenceVoid: `interface (ExecutionException) = ();`

The reference to an operation's responder is voided.

ResponderMissing: `interface (ExecutionException) = ();`

An operation's responder is missing.

ResponderNil: `interface (ExecutionException) = ();`

A feature is undefined or private and the responder is nil.

ResultInvalid: `interface (ExecutionException) = ();`

An operation's result doesn't satisfy its type.

ResultMissing: `interface (ExecutionException) = ();`

An operation's result is missing.

VariableUndefined: `interface (ExecutionException) = ();`

A local variable identifier is undefined.

Existing Connection Means

Object
• Means
• • **Existing Connection Means** (Reservable Means)

Class

```
ExistingConnectionMeans: interface
   (Means, ReservableMeans) =
   (
      public
         see connectionID, initialize
   );
```

A reservable means that involves a previously, but not necessarily presently, established connection between two engines.

Constructor

```
initialize: op ()
   throws FeatureUnavailable;
```

Exception Throws `FeatureUnavailable` if the feature is requested.

Attributes

```
connectionID: OctetString;
```

The connection identifier of the connection that the responder involves.

Exit Event

Object
• Event
• • Process Event
• • • **Exit Event**

Class

```
ExitEvent: interface (ProcessEvent) =
   (
      public
         see initialize, record
   );
```

A process event that involves the exit of a process from a place.

Constructor

initialize: op (source: Telename|Nil; time: Time|Nil);

Sets the responder's attributes according to the following table.

Attribute	*Argument*	*Default*
source	Yes	nil
time	Yes	nil
record	–	nil

Attributes

record: Object;

A record of the process whose exit the responder involves. The record is that maintained by the place the process has exited.

Identifier

Object
• Primitive (Protected)
• • **Identifier** (Ordered)

Class

```
Identifier: sealed interface (Primitive, Ordered) =
   (
      public
         see asString
   );
```

A primitive that distinguishes one object from others.

Specializations.

- The `copy` operation regards the responder's text as its only property.
- The `order` operation relates two identifiers as it does their texts.

Conversions

```
asString: op () String;
```

Returns a copy of the responder's text.

Integer

Object
• Primitive (Protected)
• • Number (Ordered)
• • • **Integer**

Class

```
Integer: sealed interface (Number) =
  (
    public
      see modulus, quotient
  );
```

A number whose values are mathematical integers.

Specializations.

- The `asCharacter` operation returns the character whose Unicode value is the responder. If the latter isn't in the interval [0, 65535], throws Conversion Unavailable.
- The `asInteger` operation returns the responder.
- The `asOctet` operation returns the responder's unsigned encoding. Bit i encodes 2^i. If the latter isn't in the interval [0, 255], throws Conversion Unavailable.
- The `asOctetString` operation returns the responder's twos complement encoding. The encoding's length is the smallest integer, n, that places the responder in the open interval $[-2^{8n-1}, 2^{8n-1})$. The encoding's bits are as follows. Bit 7 of the octet at position 1 encodes -2^{8n-1}. Any other Bit j of the octet at position i encodes $2^{8(n-i)+j}$.
- The `asReal` operation returns the real that arithmetically equals the responder.
- The `asString` operation returns a string that would represent the responder in a character telescript.
- The `copy` operation regards the responder's value as its only property.
- The `difference`, `multiply`, or `sum` operation returns an instance of the argument's class.
- The `divide` operation returns a real.
- The `magnitude` or `negate` operation returns an integer.

Operations

```
modulus: op (divisor: Integer) Integer
    throws DivisionByZero;
```

Returns the arithmetic remainder of the responder, the dividend, and the `divisor` argument, the divisor. The sign of the result is that of the dividend.

Note This operation isn't defined as it is defined in number theory.

Exception Throws `DivisionByZero` if the divisor is 0.

```
quotient: op (divisor: Integer) Integer
    throws DivisionByZero;
```

Returns the arithmetic quotient of the responder, the dividend, and the `divisor` argument, the divisor.

Exception Throws `DivisionByZero` if the divisor is 0.

Iterator

Object
• **Iterator**

Class

```
Iterator: abstract interface[Item: Class] =
   (
      public
         see current, isDone, next
   );
```

An object that *produces* zero or more objects, the iterator's *items*.

This is a class family. The `Item` parameter is the required interface member class of each item of every member of a derived class.

Note. A nil item makes the `current` and `next` attributes difficult to interpret.

Attributes

```
current: abstract readonly Item|Nil;
```

The item the responder produced last. If the responder has produced no items or has been asked to produce another after producing them all, nil.

```
isDone: abstract readonly Boolean;
```

True if the responder has produced all of its items.

```
next: abstract readonly Item|Nil
   throws ReferenceProtected;
```

The item the responder produces next. Each act of getting this attribute produces another item. If the responder has produced all of its items, nil.

Exception. Throws `ReferenceProtected` if the responder is protected.

Kernel Exception

Object
- Exception
  - Programming Exception
    - **Kernel Exception**

Class

```
KernelExeption: abstract interface
   (ProgrammingException) = ();
```

A programming exception in the processing of an object or in execution.

Subclasses

```
ClassAbstract: interface (KernelException) = ();
```

A new object is requested of an abstract class.

```
ConversionUnavailable: interface (KernelException) = ();
```

A requested conversion is undefined.

```
EncodingInvalid: interface (KernelException) = ();
```

An encoding of an object is invalid.

```
EncodingUnavailable: interface (KernelException) = ();
```

An object can’t be encoded because it is an uncopied object.

```
MarkMissing: interface (KernelException) = ();
```

A stack’s items don’t include a mark.

```
ObjectFrozen: interface (KernelException) = ();
```

An object can’t be used because it is frozen.

```
ObjectUninitialized: interface (KernelException) = ();
```

A method for the constructor violates a restriction to which it is subject.

```
ObjectUnowned: interface (KernelException) = ();
```

An object is manipulated by a process other than its owner.

List

Object
- Collection (Compared)
- • **List** (Ordered)

Class

```
List: interface[Item: Class; Match: Class<:Compared]
   (Collection[Item, Match], Ordered) =
   (
      public
         see add, append, drop, find, get, reposition, set, splice,
            transpose
   );
```

A collection whose items are associated one-to-one with the integers in the interval [1, *n*], the items' *positions*, where *n* is the collection's length.

This is a class family. The `Item` parameter is the required interface member class of each item of every member of a derived class. The `Match` parameter is the interface member class required to decide whether two items *match*.

A *sublist* is zero or more items at successive positions in a list. A sublist is sometimes defined by an operation's `initialPosition` (P_1) and `beyondFinalPosition` (P_2) arguments. The sublist consists of the items at positions in the open interval [P_1, P_2). Nil signifies that P_1 is 1 or that P_2 is n+1, where n is the list's length. The open interval [1, 1) consists of the zero items before the item, if any, at position 1. The open interval [n+1, n+1) consists of the zero items after the item, if any, at position n. If P_1 or P_2 isn't in the interval [1, n+1] or P_2 is less than P_1, the interval is malformed.

Specializations.

- The constructor makes the signature positions of its arguments the positions of the corresponding items in the responder.
- The `asList` operation positions the items in the result as in the responder.
- The `copy` operation regards the responder's items as properties. Two lists are copy-equal if their like-positioned items are copy-equal.
- The `exclude` operation decreases by one the positions of items positioned after the excluded item.
- The `include` operation positions the included item at the responder's new length.
- The `iterator` operation produces the responder's items in order of increasing position.
- The `order` operation considers the like-positioned items of two lists in order of increasing position. If the relationship between two items isn't `equal`, the lists are related or unrelated as those two items are. Otherwise if the lists are equal in length, the lists themselves are equal. Otherwise the shorter list is before the longer, and the longer is after the shorter.
- The `shallowCopy` operation preserves the order of the responder's items.

Operations (modification)

add: op (position: Integer; item: Item)
throws PositionInvalid, ReferenceProtected;

Includes the `item` argument in the responder as a new item whose position is the `position` argument. The operation increases by one the position of each item whose position is equal to or after that of the included item.

Exceptions.

- Throws `PositionInvalid` if the position is invalid.
- Throws `ReferenceProtected` if the responder is protected.

---

append: op (items: List[Item, Match])
throws ReferenceProtected;

Includes the items of the `items` argument in the responder as new items whose positions exceed the responder's length, when the operation is requested, by their argument positions. Leaves the argument unchanged.

Exception Throws `ReferenceProtected` if the responder is protected.

---

drop: op (position: Integer) Item
throws PositionInvalid, ReferenceProtected;

Excludes from the responder and returns the item whose position is the `position` argument. The operation decreases by one the position of each item whose position is after that of the excluded item.

Exceptions.

- Throws `PositionInvalid` if the position is invalid.
- Throws `ReferenceProtected` if the responder is protected.

---

reposition: op (currentPosition, newPosition: Integer)
throws PositionInvalid, ReferenceProtected;

Moves an item of the responder from one position to another, adjusting the positions of other items as required. The `currentPosition` argument gives the item's existing position, the `newPosition` argument its new position.

Note Both positions are interpreted before the responder is modified.

Exceptions.

- Throws `PositionInvalid` if either position is invalid.
- Throws `ReferenceProtected` if the responder is protected.

```
set: op (position: Integer; item: Item)
   throws PositionInvalid, ReferenceProtected;
```

Includes the `item` argument in the responder as a new or replacement item whose position is the `position` argument. If an item occupies that position in the responder already, the operation first excludes and discards that item. Otherwise the position is the responder's length and the item is appended.

Exceptions.

- Throws `PositionInvalid` if the position is invalid.
- Throws `ReferenceProtected` if the responder is protected.

```
splice: op (
   initialPosition, beyondFinalPosition: Integer|Nil;
   items: List[Item, Match]|Nil) List[Item, Match]|Nil
      throws PositionInvalid, ReferenceProtected;
```

First constructs an empty list as a candidate result. If the result remains empty after the processing described next, the operation returns nil instead.

Processes certain items of the responder, in order of decreasing position, by dropping them from the responder and adding them to the result at position 1. The items, a sublist, are those at positions in the open interval [*I*, *F*). *I* is the `initialPosition` argument, *F* the `beyondFinalPosition` argument.

If the `items` argument isn't nil, the operation continues by processing all items of the argument, in order of decreasing position, by adding them to the responder at position *I*. The operation leaves the argument unchanged.

Exceptions.

- Throws `PositionInvalid` if the sublist interval is malformed.
- Throws `ReferenceProtected` if the responder is protected.

```
transpose: op (position1, position2: Integer)
   throws PositionInvalid, ReferenceProtected;
```

Interchanges the items of the responder at two positions. Leaves the positions of other items unchanged. The `position1` argument gives one item's existing position, the `position2` argument the other's.

Exceptions.

- Throws `PositionInvalid` if either position is invalid.
- Throws `ReferenceProtected` if the responder is protected.

Operations (examination)

```
find: op (initialPosition: Integer; item: protected Item)
   Integer /* positive */ |Nil
      throws PositionInvalid;
```

Returns the position in the responder of the item that both matches the `item` argument and is at a position neither before the `initialPosition` argument nor after that of another item that satisfies the first two criteria. Leaves the responder unchanged. If there is no such item, returns nil.

Exception Throws `PositionInvalid` if the position is invalid.

```
get: op (position: Integer) Item
   throws PositionInvalid;
```

Returns the item of the responder whose position is the `position` argument. Leaves the responder unchanged.

Exception Throws `PositionInvalid` if the position is invalid.

Means

Object
• **Means**

Class

```
Means: abstract interface = ();
```

An object that identifies a means of communication (for example, the public switched telephone network) (for example, for beginning a trip).

Meeting Agent

Meeting Agent

Class

```
MeetingAgent: mixin interface (Agent) =
   (
      public
         see meeting
   );
```

An agent that plays the role of petitionee in a meeting that a meeting place arranges. Provided that it doesn't prompt the event itself, a meeting agent senses a meeting's beginning by providing a method for the `meeting` operation, and senses its end by receiving a signal, a member of Part Event.

Specialization. The `meeting` operation returns nil.

Operations

```
meeting: sponsored op (
   agent: protected Telename /* assigned */ ;
   _class: protected ClassName!;
   petition: protected Petition) Object|Nil
      throws FeatureUnavailable, MeetingDenied;
```

Requested by the engine with the desired priority of the petitioner whose petition is the `petition` argument, whose name is the `agent` argument, and whose interface member classes include the one the `class` argument denotes. The result is the responder's record of the meeting.

If the operation throws any exception other than Meeting Denied, the engine throws Meeting Denied in its place.

Exceptions.

- Throws `FeatureUnavailable` if the requester is not the engine. The engine, not a method for the operation, throws the exception.
- Throws `MeetingDenied` if the meeting is denied.

Meeting Exception

Object
• Exception
• • **Meeting Exception**

Class

`MeetingException:` `abstract interface (Exception) = ();`

An exception thrown when a meeting fails.

Subclasses

`MeetingDenied:` `interface (MeetingException) = ();`

The petitioner is denied a meeting.

`MeetingDuplicated:` `interface (MeetingException) = ();`

The petitioner and the petitionee are the same agent or are already meeting.

`PetitionExpired:` `interface (MeetingException) = ();`

The petitioner can't meet the petitionee in the maximum time allowed.

Meeting Place

Meeting Place

Class

```
MeetingPlace: mixin interface (Place) =
   (
      public
         see meet, part, partAll
   );
```

A place that uses a certain protocol, embodied by the features native to Meeting Place and Meeting Agent, to arrange meetings between occupants.

Operations

```
meet: op (
   petition: protected Petition;
   record: Object|Nil) Agent
      throws MeetingException, PlaceNotCurrent,
         ProcessNotControlled, ReferenceProtected;
```

Arranges the meeting that the `petition` argument defines between the current sponsor, the petitioner, and the operation's result, the petitionee. The `record` argument is registered as the petitioner's record of the meeting.

Exceptions.

- Throws `MeetingException` if the meeting fails.
- Throws `PlaceNotCurrent` if the current sponsor doesn't occupy the responder.
- Throws `ProcessNotControlled` unless the current sponsor, an agent, is the responder not only for either the sponsored operation that the current process performs at the engine's request or one of the operations it entails, but also for all operations entailed by that operation.
- Throws `ReferenceProtected` if the responder is protected.

```
part: op (agent: Telename)
   throws PlaceNotCurrent, ProcessNotControlled,
      ReferenceProtected;
```

Ends any meetings between the current sponsor and any agents that the `agent` argument identifies.

Exceptions.

- Throws `PlaceNotCurrent` if the current sponsor doesn't occupy the responder.
- Throws `ProcessNotControlled` unless the current sponsor, an agent, is the responder not only for either the sponsored operation that the current process performs at the engine's request or one of the operations it entails, but also for all operations entailed by that operation.
- Throws `ReferenceProtected` if the responder is protected.

```
partAll: op ()
   throws PlaceNotCurrent, ProcessNotControlled,
      ReferenceProtected;
```

Ends all meetings that involve the current sponsor, and isolates the sponsor.

Exceptions.

- Throws `PlaceNotCurrent` if the current sponsor doesn't occupy the responder.
- Throws `ProcessNotControlled` unless the current sponsor, an agent, is the responder not only for either the sponsored operation that the current process performs at the engine's request or one of the operations it entails, but also for all operations entailed by that operation.
- Throws `ReferenceProtected` if the responder is protected.

Miscellaneous Exception

Object
• Exception
• • Programming Exception
• • • **Miscellaneous Exception**

Class

```
MiscellaneousException: abstract interface
   (ProgrammingException) = ();
```

A programming exception thrown by a calendar time or a pattern.

Subclasses

```
PatternInvalid: interface (MiscellaneousException) = ();
```

A pattern’s proposed text is syntactically in error.

Named

Named

Class

```
Named: sealed mixin interface =
   (
      public
         see initialize, name
   );
```

An object that has an assigned telename.

Constructor

```
initialize: op (identity: owned OctetString|Nil);
```

Sets the responder's attributes according to the following table.

Attribute	*Argument*	*Default*
name	Yes. Sets the identity attribute to the identity argument.	Defaults the identity argument to a newly assigned octet string. Sets the authority attribute as specified by Package and Process.

Attributes

```
name: sealed readonly protected Telename /* assigned */ ;
```

The responder's telename.

Nil

Object
- Primitive (Protected)
  - **Nil**

Class

```
Nil: sealed interface (Primitive) = ();
```

A lightweight primitive with one possible value, *nil*. Nil is used to indicate the absence of an instance of another class.

Number

Object
• Primitive (Protected)
• • **Number** (Ordered)

Class

```
Number: abstract interface (Primitive, Ordered) =
   (
      public
         see asCharacter, asInteger, asOctet, asOctetString, asReal,
            asString, ceiling, difference, divide, floor, magnitude,
            multiply, negate, round, sum
   );
```

A lightweight primitive that can perform basic arithmetic operations.

Specialization. The `order` operation relates two numbers mathematically.

Operations (binary)

```
difference: abstract op (subtrahend: Number) Number;
```

Returns the arithmetic difference between the responder, the minuend; and the `subtrahend` argument, the subtrahend.

```
divide: abstract op (divisor: Number) Number
   throws DivisionByZero;
```

Returns the arithmetic quotient of the responder, the dividend; and the `divisor` argument, the divisor.

Exception. Throws `DivisionByZero` if the divisor is 0.

```
multiply: abstract op (number: Number) Number;
```

Returns the arithmetic product of the responder and the `number` argument.

```
sum: abstract op (number: Number) Number;
```

Returns the arithmetic sum of the responder and the `number` argument.

Operations (unary)

ceiling: `abstract op () Integer;`

Returns the smallest integer not arithmetically less than the responder.

---

floor: `abstract op () Integer;`

Returns the largest integer not arithmetically greater than the responder.

---

magnitude: `abstract op () Number;`

Returns the responder's absolute value.

---

negate: `abstract op () Number;`

Returns the responder's arithmetic negative.

---

round: `abstract op () Integer;`

Returns the integer that is arithmetically nearest to the responder. If two integers are equally near, which one the operation returns is undefined.

Conversions

asCharacter: `abstract op () Character`
`throws ConversionUnavailable;`

Returns a character that reflects the responder.

Exception Throws `ConversionUnavailable` if there is no such character.

---

asInteger: `abstract op () Integer;`

Returns an integer that reflects the responder.

---

asOctet: `abstract op () Octet`
`throws ConversionUnavailable;`

Returns an octet that reflects the responder.

Exception Throws `ConversionUnavailable` if there is no such octet.

---

asOctetString: `abstract op () OctetString;`

Returns an octet string that reflects the responder.

---

asReal: `abstract op () Real;`

Returns a real that reflects the responder.

---

asString: `abstract op () String;`

Returns a string that reflects the responder.

Object

Object

Class

```
Object: abstract interface =
   (
      public
         see class, copy, encode, initialize, isEqual, isFrozen,
            isInstance, isInstanceByName, isLocked, isMember,
            isMemberByName, isolate, isOwned, isProtected, isSame,
            lock, protect, size, unlockedCopy
      private
         see owner
   );
```

A unit of information and information processing.

Specialization. The `copy` operation doesn't consider the responder's `isOwned`, `isProtected`, or `owner` attribute to be among its properties. The copy's `isOwned` and `owner` attributes reflect the copy's owner. If the responder and thus the copy are protected objects, the `isProtected` attribute of the copy is *true*.

Constructor

initialize: op ();

Sets the responder's attributes according to the following table.

Attribute	*Argument*	*Default*
class	–	As required
isFrozen	–	*false*
isLocked	–	If a protected object, *true*
isOwned	–	*true*
isProtected	–	If a protected object, *true*
owner	–	As required
size	–	As required

Attributes (booleans)

isFrozen: sealed readonly Boolean;

True if the responder is frozen.

isLocked: `sealed readonly Boolean;`

True if the responder is locked.

---

isOwned: `sealed readonly Boolean;`

True if the current owner owns the responder.

---

isProtected: `sealed readonly Boolean;`

True if the reference to the responder is protected.

Attributes (other)

class: `sealed readonly Class;`

The responder's class. However, if the responder is Class, so is this attribute.

---

owner: `sealed readonly Process;`

The responder's owner.

---

size: `sealed readonly Integer` */* nonnegative */* `;`

The responder's size in octets.

Operations (booleans)

isInstance: `sealed op (_class: Class) Boolean;`

Returns *true* if the responder is an instance of the `class` argument.

---

isInstanceByName: `sealed op (_class: protected ClassName) Boolean;`

Returns *true* if the responder is an instance of the `class` argument's subject.

---

isMember: `sealed op (_class: Class) Boolean;`

Returns *true* if the responder is an interface member of the `class` argument.

`isMemberByName``: sealed op (_class: protected ClassName) Boolean;`

Returns *true* if the responder is an interface member of the `class` argument's subject.

Operations (referencing)

`isSame``: sealed op (object: protected Object) Boolean;`

Returns *true* if the `object` argument is the responder.

`protect``: sealed op () protected Object;`

Returns a protected reference to the responder.

Operations (copying)

`copy``: sealed op () Object;`

Returns a copy of the responder.

`isEqual``: sealed op (object: protected Object) Boolean;`

Returns *true* if the `object` argument is the responder or is copy-equal to it.

Operations (locking)

`lock``: sealed op () throws ObjectUnowned, ReferenceProtected;`

Locks the responder. If it is locked already, the operation has no effect.

Exceptions.

- Throws `ObjectUnowned` if the current owner doesn't own the responder.
- Throws `ReferenceProtected` if the responder isn't locked but is protected.

`unlockedCopy``: sealed op () Object;`

Returns an unlocked copy of the responder.

Operations (other)

```
encode: sealed op (
   packages: protected Set[Telename, Equal]|Nil;
   isNotInPackages: Boolean|Nil) OctetString
      throws EncodingUnavailable;
```

Returns an octet string that encodes a copy of the responder according to the encoding rules. The encoding may omit any objects in the copy's closure for which the packages that the `packages` and `isNotInPackages` arguments select provide replacements (see "Searching packages for objects").

Exception Throws `EncodingUnavailable` if the responder is an uncopied object.

```
isolate: sealed op ()
   throws ObjectUnowned, ReferenceProtected;
```

Isolates the responder.

Exceptions.

- Throws `ObjectUnowned` if the current owner doesn't own the responder.
- Throws `ReferenceProtected` if the responder is protected.

Octet

Object
• Primitive (Protected)
• • **Octet** (Ordered)

Class

```
Octet: sealed interface (Primitive, Ordered) =
   (
      public
         see asInteger
   );
```

A lightweight primitive each of whose values consists of 8 bits. For reference purposes, the bits are designated Bit 7 through Bit 0. A *zero* is 00_{16}.

Specializations.

- The copy operation regards the responder's value as its only property.
- The order operation relates two octets as it does the two lists that consist of the bits of the two octets, arranged so that Bit *i* of an octet is in position (7–*i*+1) of a list.

Conversions

```
asInteger: op () Integer /* [0, 255] */ ;
```

Returns the integer whose unsigned encoding is the responder. Bit *i* of the encoding represents 2^i.

Octet String

Object
• Collection (Compared)
• • List (Ordered)
• • • **Octet String**

Class

```
OctetString: sealed interface (List[Octet, Equal]) =
  (
     public
        see asBitString, asInteger, asString, decode, initialize
  );
```

A list whose items are octets.

Specialization. The `order` operation relates two octet strings as though first enough zeros were prepended to one octet string to make it equal in length to the other. An octet string is unrelated to an instance of any other class.

Constructor

```
initialize: op (
   octets: protected Object /* Octet|OctetString */  ...);
```

Sets the responder's items to the octets the arguments contribute. The positions of the items reflect their order of contribution. An octet contributes itself. An octet string contributes its items in order of increasing position. The arguments contribute in order of increasing signature position.

Operations

```
decode: sealed op (
   packages: protected Set[Telename, Equal]|Nil;
   isNotInPackages: Boolean|Nil) Object
      throws ClassUnavailable, EncodingInvalid;
```

Returns the object that the responder encodes according to the encoding rules. The encoding may omit any objects in the object's closure for which the packages that the `packages` and `isNotInPackages` arguments select provide replacements (see "Searching packages for objects").

Exceptions.

- Throws `ClassUnavailable` if the encoding omits an object for which none of the packages provides a replacement.
- Throws `EncodingInvalid` if the encoding is invalid.

Conversions

asBitString: `op () BitString;`

Returns a bit string whose length is 8 *n*. *n* is the responder's length. The bit at position (8 *i*+*j*) is Bit (8– *j*) of the octet at position *i*. In the above, *i* and *j* are integers, *i* is in the open interval [0, *n*), and *j* is in the interval [1, 8].

---

asInteger: `op () Integer;`

Returns the integer whose method for the `asOctetString` operation returns the responder.

---

asString: `op () String throws ConversionUnavailable;`

Returns a string whose characters the responder's octets would represent in a binary telescript (see String's `asOctetString` operation).

Exception Throws `ConversionUnavailable` if the responder is malformed.

Ordered

Ordered

Class

```
Ordered: mixin interface =
   (
      public
         see maximum, minimum, order
   );
```

One of a set of objects that are partially ordered. Any object in the set is `before`, `equal to`, `after`, or `unordered` with respect to any other of the objects. The indicated identifiers denote the four possible relationships.

Two objects are unordered if either or both aren't ordered objects. Also, two members of different immediate subclasses of Ordered are unordered.

Operations

```
maximum: op (object: Ordered) Ordered;
```

If the responder is after the `object` argument, returns the responder. Otherwise returns the object.

```
minimum: op (object: Ordered) Ordered;
```

If the responder is before the `object` argument, returns the responder. Otherwise returns the object.

```
order: abstract op (object: protected Ordered)
   Identifier /* as enumerated */ ;
```

Returns the identifier that relates the responder to the `object` argument. For example, if the responder is before the object, the identifier is `before`.

Package

Object
- Collection (Compared)
- • Set (Verified)
- • • Dictionary
- • • • **Package** (Named and Protected)

Class

```
Package: interface[Key, Value: Class; Match: Class<:Compared]
   (Dictionary[Key, Value, Match], Named, Protected) =
   (
      public
         see compatibles, initialize
   );
```

A dictionary that is protected and thus locked, whose keys and values are locked, that is denoted by an assigned telename, and that may assert backward compatibility with certain other packages of the same authority.

This is a class family. The `Key` parameter is the required interface member class of each key of every member of a derived class. The `Value` parameter is the required interface member class of each value. The `Match` parameter is the interface member class required to decide whether two keys *match*. Equal decides whether two values *match*.

Specializations.

- The `examine`, `find`, or `get` operation effectively passes its result *byCopy*.
- The `iterator` operation's result passes its `current` or `next` attribute *byCopy*.

Constructor

```
initialize: op (
   identity: owned OctetString|Nil;
   compatibles: owned Set[OctetString, Equal]|Nil;
   keysAndValues: owned Object ...
      /* (even numbered) key: Key; value: Value */);
```

Sets the responder's attributes according to the following table. Sets the responder's keys and values to the arguments. Locks them before adding them, using the `set` operation, in order of increasing signature position.

Attribute	*Argument*	*Default*
`name`	Yes. Sets the `identity` attribute to the `identity` argument.	Defaults the `identity` argument to a newly assigned octet string. Sets the `authority` attribute to that of the `name` attribute of the current owner.
`compatibles`	Yes	empty

Attributes

`compatibles:` `sealed readonly Set[OctetString, Equal];`

The `identity` attributes that distinguish, from the responder's `name` attribute, the `name` attributes of packages assertedly forward compatible with the responder. The language leaves "forward compatible" undefined.

Package Process

Package Process

Class

```
PackageProcess: sealed mixin interface =
   (
      public
         see find, findClass, findPackage, get, getPackage,
            initialize
      private
         see freeze, privatePackages, publicPackages, thaw
   );
```

A process that offers packages.

Constructor

```
initialize: op ();
```

Sets the responder's attributes according to the following table.

Attribute	*Argument*	*Default*
privatePackages	–	empty
publicPackages	–	empty

Attributes

```
privatePackages: sealed readonly List[Package, Equal];
```

The responder's private packages.

```
publicPackages: sealed readonly List[Package, Equal];
```

The responder's public packages.

Operations (packages)

```
findPackage: sealed op (value: protected Object)
   protected Telename /* assigned */ |Nil;
```

Returns the telename of the package that the `find` operation would choose if the `value` argument were its first argument and nil were its second. If no package would be chosen, returns nil.

```
getPackage: sealed op (key: protected Object)
   protected Telename // assigned
      throws KeyInvalid;
```

Returns the telename of the package that the `get` operation would choose if the `key` argument were its first argument and nil were its second.

Exception Throws `KeyInvalid` if no package would be chosen.

Operations (searching)

```
find: sealed op (
   value: protected Object;
   packages: protected Telename|Nil) copied Object|Nil;
```

Returns any key whose value matches the `value` argument. The operation searches the packages that the `packages` argument selects (see “Searching packages for objects”). If there is no such key, the operation returns nil.

```
findClass: sealed op (_class: protected ClassName) Class|Nil;
```

Returns the `class` argument’s subject.

To find the class, the operation follows the class search algorithm using packages that the responder offers the requester’s owner. If the algorithm fails to produce the class, the operation returns nil.

```
get: sealed op (
   key: protected Object;
   packages: protected Telename|Nil) copied Object
      throws KeyInvalid;
```

Returns any value whose key matches the `key` argument. The operation searches the packages that the `packages` argument selects (see “Searching packages for objects”).

Exception Throws `KeyInvalid` if there is no such key.

Operations (freezing and thawing)

```
freeze: sealed op (
   object: Object;
   packages: protected Set[Telename, Equal]|Nil;
   isNotInPackages: Boolean|Nil);
```

Freezes the `object` argument or other objects in its closure. If the current owner doesn't own the argument, the operation has no effect. Otherwise if the argument is locked and is represented by a value in one of the packages that the `packages` and `isNotInPackages` arguments select (see "Searching packages for objects"), the operation freezes the argument. Otherwise for each property of the object (including its class) the responder performs the operation again after making the property the operation's `object` argument.

```
thaw: sealed op (
   object: Object;
   packages: protected Set[Telename, Equal]|Nil;
   isNotInPackages: Boolean|Nil);
```

Thaws the `object` argument or other objects in its closure. If the current owner doesn't own the argument, the operation has no effect. Otherwise if the argument is frozen and is represented by a value in one of the packages that the `packages` and `isNotInPackages` arguments select (see "Searching packages for objects"), the operation thaws the argument. Otherwise for each property of the object (including its class) the responder performs the operation again after making the property the operation's `object` argument.

Part Event

Object
• Event
• • Process Event
• • • **Part Event**

Class

```
PartEvent: interface (ProcessEvent) =
  (
    public
      see initialize, record
  );
```

A process event that involves the parting of two agents.

Constructor

```
initialize: op (source: Telename|Nil; time: Time|Nil);
```

Sets the responder's attributes according to the following table.

Attribute	*Argument*	*Default*
source	Yes	nil
time	Yes	nil
record	–	nil

Attributes

```
record: Object;
```

One agent's record of its acquaintance with another. The latter agent is the one that initiated the parting that the responder involves.

Pattern

Object
- **Pattern**

Class

```
Pattern: interface =
   (
      public
         see find, initialize, split, substitute
   );
```

An object able to lexically analyze and modify strings.

Specialization. The copy operation the responder's text as its only property.

Constructor

```
initialize: op (text: copied String)
   throws PatternInvalid;
```

Sets the responder's text to the argument.

Exception Throws PatternInvalid if the text is invalid.

Operations

```
find: op (
   subject: protected String;
   initialPosition, beyondFinalPosition: Integer|Nil)
      List[Integer, Equal] /* two positive integers */ |Nil
         throws PositionInvalid;
```

Searches the subject argument for the first substring that matches the responder. The search is confined to the substring of the argument that consists of the items at positions in the open interval [I, F). I is the initialPosition argument, F the beyondFinalPosition argument.

If it finds a matching substring, the operation returns the integers, r_1 and r_2, such that the open interval [r_1, r_2) defines the matching substring. Otherwise it returns nil.

Exception Throws PositionInvalid if the subject's interval is malformed.

```
split: op (
   subject: protected String;
   includeMatches: Boolean;
   repetitions: Integer /* nonnegative */ |Nil)
      List[String, Equal];
```

Searches the `subject` argument for the first substrings that match the responder. If there are no matches, the operation returns a copy of the string.

If the `repetitions` argument isn't nil, the operation finds at most the number of substrings the argument specifies. Otherwise it finds them all.

A search that finds *n* matches thereby divides the subject into 2**n*+1 substrings. The substrings with even ordinal numbers are the matches. The others are, in order, the substring before the first match, the substrings between matches, and the substring after the last match.

The operation returns, as the items of a list, the substrings into which the search divides the subject. However, if the `includeMatches` argument is *false*, the list excludes the matching substrings themselves.

```
substitute: op (
   subject: unprotected String;
   replacement: protected String;
   repetitions: Integer /* nonnegative */ |Nil)
      Integer /* nonnegative */ ;
```

Searches the `subject` argument for the first substrings that match the responder and returns the number it finds.

If the `repetitions` argument isn't nil, the operation finds at most the number of substrings the argument specifies. Otherwise it finds them all.

The operation replaces each matching substring, m_i, with a modified copy of the `replacement` argument, which is modified as follows. Wherever "`&`" occurs in the copy in an escape sequence, the preceding "\" is deleted. Wherever else "`&`" occurs in the copy, the "`&`" is replaced with a copy of m_i.

Permit

Object
• **Permit**

Class

```
Permit: sealed interface =
   (
      public
         see age, authenticity, canCharge, canCreate, canDeny, canGo,
            canGrant, canRestart, canSend, charges, extent,
            initialize, intersection, priority
   );
```

An object that grants capabilities to a process, the permit's *subject*

Constructor

```
initialize: op (
   age, charges, extent: Integer /* nonnegative */ |Nil);
```

Sets the responder's attributes according to the following table.

Attribute	*Argument*	*Default*
age	Yes	nil
charges	Yes	nil
extent	Yes	nil
authenticity	–	nil
canCharge	–	*true*
canCreate	–	*true*
canDeny	–	*true*
canGo	–	*true*
canGrant	–	*true*
canRestart	–	*true*
canSend	–	*true*
priority	–	nil

Attributes (integers)

```
age: Integer /* nonnegative */ |Nil;
```

The maximum age in seconds of the responder's subject. If none, nil.

authenticity: `Integer` */* [0, 40] */* `|Nil;`

The maximum authenticity of the responder's subject. If none, nil.

---

charges: `Integer` */* nonnegative */* `|Nil;`

The maximum charges in teleclicks of the responder's subject. If none, nil.

---

extent: `Integer` */* nonnegative */* `|Nil;`

The maximum size in octets of the responder's subject. If none, nil.

---

priority: `Integer` */* [–20, 20] */* `|Nil;`

The maximum priority of the responder's subject. If none, nil.

Attributes (booleans)

canCharge: `Boolean;`

True if the responder's subject can request the `charge` operation.

---

canCreate: `Boolean;`

True if the responder's subject can request the `new` operation of a subclass of Process.

---

canDeny: `Boolean;`

True if the responder's subject can decrease the capabilities of other processes under certain circumstances (see "Receiving a permit").

---

canGo: `Boolean;`

True if the responder's subject can request the `go` operation.

---

canGrant: `Boolean;`

True if the responder's subject can increase the capabilities of other processes under certain circumstances (see "Receiving a permit").

---

canRestart: `Boolean;`

True if the responder's subject can be restarted.

`canSend``: Boolean;`

True if the responder's subject can request the `send` operation.

Operations

`intersection``: op (permit: protected Permit) Permit;`

Returns the intersection of the responder and the `permit` argument.

Note: The responder is left unchanged.

Permit Process

Permit Process

Class

```
PermitProcess: sealed mixin interface =
   (
      public
         see age, assignments, charge, charges, initialize,
            localPermit, nativePermit, permit, priority,
            regionalPermit
      private
         see desiredPriority
   );
```

A process that is subject to permits.

Constructor

```
initialize: op (
   nativePermit: copied Permit;
   localPermit: copied Permit|Nil;
   desiredPriority: Integer /* [–20, 20] */ |Nil)
      throws OccupancyDenied;
```

Sets the responder's attributes according to the following table.

Attribute	*Argument*	*Default*
nativePermit	Yes	–
localPermit	Yes	basic. See "Receiving a local permit"
desiredPriority	Yes	That of the current sponsor
age	–	0
assignments	–	0
charges	–	0
regionalPermit	–	See "Receiving a regional permit"

Exception Throws OccupancyDenied if the responder is denied occupancy.

Attributes (past)

age: `sealed readonly Integer` *// nonnegative*
`throws FeatureUnavailable;`

The responder's actual age in seconds.

Exception Throws `FeatureUnavailable` if the current sponsor isn't the responder's peer.

---

assignments: `sealed readonly Integer` *// nonnegative*
`throws FeatureUnavailable;`

The responder's actual charges in teleclicks assigned to child processes.

Exception Throws `FeatureUnavailable` if the current sponsor isn't the responder's peer.

---

charges: `sealed readonly Integer` *// nonnegative*
`throws FeatureUnavailable;`

The responder's actual charges in teleclicks.

Exception Throws `FeatureUnavailable` if the current sponsor isn't the responder's peer.

Attributes (future)

permit: `sealed readonly copied Permit;`

The responder's effective permit.

---

priority: `sealed readonly Integer` */* [–20, 20] */* `;`

The responder's actual priority.

Attributes (permits)

desiredPriority: `sealed Integer` */* [–20, 20] */* `;`

The responder's desired priority.

---

localPermit: `sealed copied Permit`
`throws FeatureUnavailable;`

The responder's local permit.

Exception Throws `FeatureUnavailable` if the current sponsor doesn't satisfy the criteria for accessing this feature (see "Receiving a local permit").

---

nativePermit: `sealed copied Permit`
`throws FeatureUnavailable;`

The responder's native permit.

Exception Throws `FeatureUnavailable` if the current sponsor doesn't satisfy the criteria for accessing this feature (see "Receiving a native permit").

---

regionalPermit: `sealed copied Permit`
`throws FeatureUnavailable;`

The responder's regional permit.

Exception Throws `FeatureUnavailable` if the current sponsor doesn't satisfy the criteria for accessing this feature (see "Receiving a regional permit").

Operations

charge: `sealed op (charges: Integer` */* nonnegative */* `)`
`throws PermitViolated;`

Increases the responder's actual charges by the `charges` argument.

Exception Throws `PermitViolated` if the current permit forbids the `charge` operation or the responder's effective permit would be exhausted as a consequence.

Petition

Object
• **Petition**

Class

```
Petition: interface =
   (
      public
         see agentClass, agentName, initialize, maximumWait
   );
```

An object that defines a meeting from the viewpoint of the petitioner. The petition's main purpose is to identify the petitionee.

Constructor

```
initialize: op (
   agentName: Telename|Nil;
   agentClass: ClassName|Nil;
   maximumWait: Integer /* nonnegative */ |Nil);
```

Sets the responder's attributes according to the following table.

Attribute	*Argument*	*Default*
agentName	Yes	nil
agentClass	Yes	nil
maximumWait	Yes	nil

Attributes

```
agentClass: ClassName|Nil;
```

An interface member class of any agent that plays the role of petitionee in the meeting that the responder defines. If no requirement is imposed, nil.

```
agentName: Telename|Nil;
```

A name for any agent that plays the role of petitionee in the meeting that the responder defines. If no requirement is imposed, nil.

```
maximumWait: Integer /* nonnegative */ |Nil;
```

The number of seconds after which the `meet` operation fails if the meeting the responder defines isn't arranged. If no requirement is imposed, nil.

Place

Object
- Process (Named, Package Process, Permit Process, and Uncopied)
- • **Place** (Unmoved)

Class

```
Place: abstract interface (Process, Unmoved) =
   (
      public
         see address, entering, initialize
   );
```

A process that other processes can occupy.

Specializations.

- The `contacts` attribute consists of unprotected references to the responder's occupants. The engine includes a reference when a process enters, excluding any such reference when the process exits.
- The `entering` operation returns nil.
- The `size` attribute excludes those of the responder's occupants.

Constructor

```
initialize: op (
   nativePermit: copied Permit;
   localPermit: copied Permit|Nil;
   desiredPriority: Integer /* [–20, 20] */ |Nil;
   location: OctetString|Nil)
      throws AddressInvalid, OccupancyDenied, PermitViolated;
```

Sets the responder's attributes according to the following table.

Attribute	*Argument*	*Default*
`nativePermit`	Yes	–
`localPermit`	Yes	Basic; see "Receiving a local permit"
`desiredPriority`	Yes	That of the current sponsor
`address`	Yes. Sets the `location` attribute to the `location` argument.	Defaults the `location` argument to that attribute of the `address` attribute, *A*, of the responder's immediate superplace. Sets the `authority` and `routingAdvice` attributes to those of *A*.
`age`	–	0
`assignments`	–	0
`charges`	–	0
`contacts`	–	empty
`isPossibleDuplicate`	–	*false*
`name`	–	Sets the `authority` attribute to that of the `name` attribute of the current sponsor, the `identity` attribute to one newly assigned.
`privatePackages`	–	empty
`publicPackages`	–	empty
`regionalData`	–	That of the current process
`regionalPermit`	–	See "Receiving a regional permit"

Exceptions.

- Throws `AddressInvalid` if the responder's teleaddress is rejected.
- Throws `OccupancyDenied` if the responder is denied occupancy.
- Throws `PermitViolated` if the current permit forbids the `new` operation or is inadequate.

Attributes

```
address: sealed readonly protected Teleaddress /* assigned */ ;
```

The responder's teleaddress.

Operations

```
entering: sponsored op (
   occupant: protected Telename /* assigned */ ;
   _class: protected ClassName!;
   permit: unprotected Permit;
   ticket: protected Ticket|Nil) Object
      throws DestinationUnknown, FeatureUnavailable,
         OccupancyDenied;
```

Requested by the engine with the desired priority of the prospective occupant whose ticket is the `ticket` argument, whose name is the `occupant` argument, and whose interface member classes include the one the `class` argument denotes. The result is the responder's record of the occupancy.

The `permit` argument is the intersection of the occupant's native, regional, and requested local permits, except that its `age` and `charges` attributes are the seconds and teleclicks that remain to the occupant, not its actual age and charges. The responder can modify the permit, thus indirectly altering the occupant's local permit to grant or deny the occupant capabilities.

The `ticket` argument is the ticket the occupant used to reach the responder, except that the engine has set its `destinationPermit` and `notes` attributes to nils. If the occupant is constructed locally, the ticket is absent.

If the operation throws a *member* of either subclass of Trip Exception listed under "Exceptions" below, the engine constructs an *instance* of that class and sets its `ticketStub` attribute authentically. If the operation throws any other exception or no trip is involved, the engine throws Occupancy Denied.

Exceptions.

- Throws `DestinationUnknown` if occupancy is denied.
- Throws `FeatureUnavailable` if the requester is not the engine. The engine, not a method for the operation, throws the exception.
- Throws `OccupancyDenied` if occupancy is denied.

Primitive

Object
- **Primitive** (Protected)

Class

```
Primitive: sealed abstract interface (Object, Protected) =
   (
      public
         see initialize
   );
```

An object that can't be constructed using the `new` operation.

Constructor

```
initialize: op ()
   throws FeatureUnavailable;
```

Exception Throws `FeatureUnavailable` if the feature is requested.

Primitive Exception

Object
• Exception
• • Programming Exception
• • • **Primitive Exception**

Class

```
PrimitiveException: abstract interface
   (ProgrammingException) = ();
```

A programming exception thrown by a primitive.

Subclasses

```
DivisionByZero: interface (PrimitiveException) = ();
```

A divisor is 0.

Process

Object
- **Process** (Named, Package Process, Permit Process, and Uncopied)

Class

```
Process: sealed abstract interface (Object, Named,
   PackageProcess, PermitProcess, Uncopied) =
   (
      public
         see initialize, isPossibleDuplicate, live, regionalData,
            wait, who
      private
         see contacts
   );
```

An object that embodies an autonomous computation.

Constructor

```
initialize: op (
   nativePermit: copied Permit;
   localPermit: copied Permit|Nil;
   desiredPriority: Integer /* [–20, 20] */ |Nil)
      throws OccupancyDenied, PermitViolated;
```

Sets the responder's attributes according to the following table.

Attribute	*Argument*	*Default*
nativePermit	Yes	–
localPermit	Yes	basic. See "Receiving a local permit"
desiredPriority	Yes	That of the current sponsor
age	–	0
assignments	–	0
charges	–	0
contacts	–	empty
isPossibleDuplicate	–	*false*
name	–	Sets the authority attribute to that of the name attribute of the current sponsor. Sets the identity attribute to a newly assigned octet string.
privatePackages	–	empty
publicPackages	–	empty
regionalData	–	That of the current process
regionalPermit	–	See "Receiving a regional permit"

Exceptions.

- Throws `OccupancyDenied` if the responder is denied occupancy.
- Throws `PermitViolated` if the current permit forbids the `new` operation or is inadequate.

Attributes

```
contacts: sealed readonly
   Dictionary[Process, Object, Equal];
```

The responder's records of processes with which the process is in contact.

Note. The engine includes processes in, and excludes processes from, this attribute as specified by Agent's and Place's specializations of it.

```
isPossibleDuplicate: sealed readonly Boolean;
```

True if the responder is a possible duplicate.

`regionalData``: sealed protected Object`
`throws FeatureUnavailable;`

The responder's brand.

Exception Throws `FeatureUnavailable` if the current sponsor doesn't satisfy the criteria for accessing this feature (see "Branding a process").

Operations

`live``: abstract sponsored op (cause: Exception|Nil)`
`throws Exception, FeatureUnavailable;`

Requested by the engine with the desired priority that the `new` operation prescribes. If the `cause` argument isn't nil, the responder is restarted after throwing that exception. Otherwise the responder is started.

Exceptions.

- Throws `Exception` if the responder wishes to be restarted.
- Throws `FeatureUnavailable` if the requester is not the engine. The engine, not a method for the operation, throws the exception.

`wait``: sealed op (seconds: Integer` */* nonnegative */* `)`
`throws ProcessNotCurrent;`

Blocks the current process for the number of seconds that the `seconds` argument specifies.

Exception Throws `ProcessNotCurrent` if the responder isn't the current process.

`who``: sealed op () Object|Nil;`

Returns the requester's owner's record of its meeting with or occupancy by the responder. If there is no such record, returns nil.

Process Event

Object
• Event
• • **Process Event**

Class

```
ProcessEvent: interface (Event) = ();
```

An event in the life of a process.

Process Exception

Object
• Exception
• • Programming Exception
• • • **Process Exception**

Class

```
ProcessException: abstract interface
   (ProgrammingException) = ();
```

A programming exception in the processing of a process.

Subclasses

```
AddressInvalid: interface (ProcessException) = ();
```

A place's assigned teleaddress is rejected, or an engine place's assigned telename and assigned teleaddress differ in authority.

```
ConditionUnavailable: interface (ProcessException) = ();
```

A resource's condition is set without the resource's use.

```
ConditionUndefined: interface (ProcessException) = ();
```

A resource's proposed condition is undefined.

```
PackageUnavailable: interface (ProcessException) = ();
```

A requested package is unavailable.

```
PermitViolated: interface (ProcessException) = ();
```

The current permit is violated.

```
PlaceNotCurrent: interface (ProcessException) = ();
```

The current sponsor doesn't occupy a feature's responder.

```
ProcessNotControlled: interface (ProcessException) = ();
```

An agent isn't in a state that allows performance of the `go`, `send`, `meet`, `part`, or `partAll` operation. (See those operations for specific limitations.)

ProcessNotCurrent: `interface (ProcessException) = ();`

A feature's responder isn't the current process.

Programming Exception

Object
• Exception
• • **Programming Exception**

Class

```
ProgrammingException: abstract interface (Exception) =
   ();
```

An exception that indicates that a programming mistake was made in the provision or use of a feature.

Protected

Protected

Class

```
Protected: mixin interface = ();
```

An object that, while unlocked throughout its performance of the constructor, is permanently locked at the moment that the `new` operation succeeds.

Specialization. The `isSame` operation returns *true* if the objects are the same or copy-equal.

Real

Object
• Primitive (Protected)
• • Number (Ordered)
• • • **Real**

Class

```
Real: sealed interface (Number) = ();
```

A number whose values are real numbers and whose behavior with respect to them conforms to *IEEE Standard for Binary Floating-Point Arithmetic*.

Specializations.

- The `asCharacter`, `asOctet`, or `asOctetString` operation behaves as would the integer the responder would return if it performed the `asInteger` operation.
- The `asInteger` operation discards the responder's fractional part so as to arithmetically truncate the responder toward 0.
- The `asReal` operation returns the responder.
- The `asString` operation returns a string that would represent the responder in a character telescript.
- The `copy` operation regards the responder's value as its only property.
- The `difference`, `divide`, `magnitude`, `multiply`, `negate`, or `sum` operation returns a real.

Reservable Means

Reservable Means

Class

```
ReservableMeans: mixin interface (Means) =
   (
      public
         see initialize, isReserved, reserve
   );
```

A means that can be reserved.

Constructor

```
initialize: op ();
```

Sets the responder's attributes according to the following table.

Attribute	*Argument*	*Default*
isReserved	–	*false*

Attributes

```
isReserved: readonly Boolean;
```

True if the responder is presently reserved.

Operations

```
reserve: op (interval: Integer /* nonnegative */ )
   throws ReservationUnavailable;
```

Reserves the responder for the interval in seconds that the `interval` argument specifies, overriding any existing reservation of the responder.

Exception Throws `ReservationUnavailable` if no reservation can be made.

Resource

Object
• **Resource**

Class

```
Resource: interface =
   (
      public
         see condition, conditions, initialize
   );
```

An object that can control its concurrent use by multiple processes.

Specialization. The copy operation behaves as though neither uses nor awaited uses of the responder affect its properties. Thus no processes initially use, or await use of, the result of the copy operation.

Constructor

```
initialize: op (
   condition: Identifier|Nil;
   conditions: copied Set[Identifier, Equal]|Nil)
      throws ConditionUndefined;
```

Sets the responder's attributes according to the following table.

Attribute	*Argument*	*Default*
condition	Yes	An undefined identifier
conditions	Yes	A set that consists of that identifier

Exception. Throws ConditionUndefined if the condition argument doesn't match an item of the conditions argument.

Attributes

```
condition: sealed Identifier
   throws ConditionUnavailable, ConditionUndefined;
```

The responder's present condition.

Exceptions.

- Throws ConditionUnavailable if the attribute is being set and either the current process doesn't have use of the responder or the use is shared.
- Throws ConditionUndefined if the attribute as set wouldn't match an item of the conditions attribute.

```
conditions: sealed readonly
   protected Set[Identifier, Equal];
```

The responder's possible conditions.

Same

Compared
- **Same**

Class

```
Same: mixin interface (Compared) =
   (
      public class
         see compare
   );
```

A compared object that matches objects that are one and the same.

Sealings

```
compare: sealed;
```

Returns `equal` if the objects are the same. Otherwise returns `unordered`.

Set

Object
• Collection (Compared)
• • **Set** (Verified)

Class

```
Set: interface[Item: Class; Match: Class<:Compared]
   (Collection[Item, Match], Verified) =
   (
      public
         see difference, intersection, union
   );
```

A collection no two of whose items match upon their inclusion.

This is a class family. The `Item` parameter is the required interface member class of each item of every member of a derived class. The `Match` parameter is the interface member class required to decide whether two items *match*.

Note. The modification of one item *in situ* can cause it to match another. In that case, the set is considered internally inconsistent.

Specializations.

- The constructor uses the `include` operation to include its arguments in order of increasing signature position.
- The `include` operation first excludes from the responder and discards any item that matches the included item.
- The `verify` operation requires that no two items match.

Operations

```
difference: op (set: protected Set[Item, Match])
   throws ReferenceProtected;
```

Excludes from the responder and discards every item that matches an item of the `set` argument. However, if the responder or the set is internally inconsistent, the operation's behavior is undefined.

Exception. Throws `ReferenceProtected` if the responder is protected.

```
intersection: op (set: protected Set[Item, Match])
   throws ReferenceProtected;
```

Excludes from the responder and discards every item that doesn't match an item of the `set` argument. However, if the responder or the set is internally inconsistent, the operation's behavior is undefined.

Exception. Throws `ReferenceProtected` if the responder is protected.

union: op (set: protected Set[Item, Match])
throws ReferenceProtected;

Includes in the responder as a new item each item of the `set` argument that doesn't match an existing item of the responder. However, if the responder or the set is internally inconsistent, the operation's behavior is undefined.

Exception Throws `ReferenceProtected` if the responder is protected.

Stack

Object
• Collection (Compared)
• • List (Ordered)
• • • **Stack**

Class

```
Stack: interface[Item: Class; Match: Class<:Compared]
   (List[Item, Match]) =
   (
      public
         see pop, push, pushItems, roll, swap
   );
```

A list that can be manipulated last-in first-out. The stack's *top* is position 1. The stack's *bottom* is the stack's length.

This is a class family. The `Item` parameter is the required interface member class of each item of every member of a derived class. The `Match` parameter is the interface member class required to decide whether two items *match*.

Operations

```
pop: op () Item
   throws ReferenceProtected, StackDepleted;
```

Excludes from the responder and returns the item at the responder's top.

Exceptions.

- Throws `ReferenceProtected` if the responder is protected.
- Throws `StackDepleted` if the responder is empty.

```
push: op (item: Item)
   throws ReferenceProtected;
```

Includes the `item` argument in the responder as a new item at its top.

Exception Throws `ReferenceProtected` if the responder is protected.

```
pushItems: op (items: List[Item, Match])
   throws ReferenceProtected;
```

Includes the items of the `items` argument in the responder as new items at the same positions—that is, at the top. Leaves the argument unchanged.

Exception Throws `ReferenceProtected` if the responder is protected.

---

roll: `op (shifts: Integer; items: Integer` */* nonnegative */* `)`
`throws ReferenceProtected, StackDepleted;`

Shifts the items at the top of the responder. The `items` argument is the number of items, *I*, to be shifted, the absolute value of the `shifts` argument the number of positions, *P*, each item is to be shifted. If either is 0, the operation has no effect. If the latter is positive, the items are shifted toward the responder's top; if negative, they are shifted toward its bottom.

To shift the topmost *I* items one position upward is to change to *I* the position of the topmost item and decrease by one the other shifted items' positions. To shift the items one position downward is to make topmost the item at position *I* and increase by one the other shifted items' positions.

Exceptions.

- Throws `ReferenceProtected` if the responder is protected.
- Throws `StackDepleted` if the responder's length is less than *I*.

---

swap: `op ()`
`throws ReferenceProtected, StackDepleted;`

Interchanges the responder's items at positions 1 and 2. Leaves the positions of other items unchanged.

Exceptions.

- Throws `ReferenceProtected` if the responder is protected.
- Throws `StackDepleted` if the responder's length is less than 2.

String

Object
- Collection (Compared)
- • List (Ordered)
- • • **String** (Cased)

Class

```
String: sealed interface (List[Character, Equal], Cased) =
   (
      public
         see asIdentifier, asInteger, asOctetString, asReal,
            initialize, substring
   );
```

A list whose items are characters.

Specializations.

- The `isLower`, `isUpper`, `makeLower`, and `makeUpper` operations use the cases of the responder's items.
- The `makeUpper` operation, if requested of a string that includes "β", returns a string in which "β" is replaced by two occurrences of "S" as Unicode requires.
- The `order` operation relates two strings as though first enough occurrences of "NUL" were appended to one string to make it equal in length to the other. A string is unrelated to an instance of any other class.

Constructor

```
initialize: op (
   characters: protected Object /* Character|String */ ...);
```

Sets the responder's items to the characters the arguments contribute. The positions of the items reflect their order of contribution. A character contributes itself. A string contributes its items in order of increasing position. The arguments contribute in order of increasing signature position.

Operations

```
substring: op (
   initialPosition, beyondFinalPosition: Integer|Nil) String
      throws PositionInvalid;
```

Returns a string whose items are those in the substring of the responder that the open interval [*I*, *F*) defines. *I* is the `initialPosition` argument. *F* is the `beyondFinalPosition` argument. Leaves the responder unchanged.

Exception Throws `PositionInvalid` if the substring interval is malformed.

Conversions

asIdentifier: `op () Identifier;`

Returns the identifier whose text copy-equals the responder.

---

asInteger: `op () Integer`
`throws ConversionUnavailable;`

Returns the integer the responder would represent in a character telescript.

Exception Throws `ConversionUnavailable` if the responder is malformed.

---

asOctetString: `op () OctetString;`

Returns an octet string whose octets would represent the responder's characters in a binary telescript (see Octet String's `asString` operation).

---

asReal: `op () Real`
`throws ConversionUnavailable;`

Returns the real the responder would represent in a character telescript.

Exception Throws `ConversionUnavailable` if the responder is malformed.

Teleaddress

Object
- **Teleaddress**

Class

```
Teleaddress: interface =
   (
      public
         see authority, initialize, location, routingAdvice
   );
```

An object that purports to denote one advertised place or all advertised places in a given region.

Constructor

```
initialize: op (
   authority, location: owned OctetString|Nil;
   routingAdvice: owned List[OctetString, Equal]|Nil);
```

Sets the responder's attributes according to the following table.

Attribute	*Argument*	*Default*
authority	Yes	That of the address attribute of the place the current process occupies
location	Yes	nil
routingAdvice	Yes	empty

Attributes

```
authority: owned OctetString;
```

The octet string that denotes the region that the responder specifies.

```
location: owned OctetString|Nil;
```

The octet string that denotes the advertised place that the responder specifies. If the responder doesn't specify an advertised place, nil.

```
routingAdvice: owned List[OctetString, Equal];
```

The octet strings that denotes the transit regions that the responder specifies. The regions are listed in order of decreasing preference.

Telename

Object
- **Telename**

Class

```
Telename: interface =
   (
      public
         see authority, identity, initialize
   );
```

An object that purports to denote either one process, all processes of a given authority, one package, or all packages of a given authority. Whether a particular telename denotes processes or packages is known from context.

Constructor

```
initialize: op (
   authority: owned OctetString;
   identity: owned OctetString|Nil);
```

Sets the responder's attributes according to the following table.

Attribute	*Argument*	*Default*
authority	Yes	–
identity	Yes	nil

Attributes

```
authority: owned OctetString;
```

The octet string that denotes the authority that the responder specifies.

```
identity: owned OctetString|Nil;
```

The octet string that denotes the process or package that the responder specifies. If the responder doesn't specify a process or package, nil.

Ticket

Object
• **Ticket**

Class

```
Ticket: sealed interface =
   (
      public
         see desiredWait, destinationAddress, destinationClass,
            destinationName, destinationPermit, initialize,
            maximumWait, notes, wayOut
   );
```

An object that defines a trip from the viewpoint of the agent that takes the trip. The ticket's main purpose is to identify the agent's destination.

Constructor

```
initialize: op (
   destinationName: Telename|Nil;
   destinationAddress: Teleaddress|Nil;
   destinationClass: ClassName|Nil;
   maximumWait: Integer /* nonnegative */ |Nil;
   wayOut: Way|Nil;
   notes: Object|Nil);
```

Sets the responder's attributes according to the following table.

Attribute	*Argument*	*Default*
destinationName	Yes	nil
destinationAddress	Yes	nil
destinationClass	Yes	nil
maximumWait	Yes	nil
wayOut	Yes	nil
notes	Yes	nil
desiredWait	–	nil
destinationPermit	–	nil

Attributes (destination)

```
destinationAddress: Teleaddress|Nil;
```

An address for the destination of the trip that the responder defines. If no requirement is imposed, nil.

destinationClass: `ClassName|Nil;`

An interface member class of the destination of the trip that the responder defines. If no requirement is imposed, nil.

---

destinationName: `Telename|Nil;`

A name for the destination of the trip that the responder defines. If no requirement is imposed, nil.

---

destinationPermit: `Permit|Nil;`

The local and regional permit that the agent requires at the destination of the trip that the responder defines. If no requirement is imposed, nil.

Attributes (waiting)

desiredWait: `Integer` */* nonnegative */* `|Nil;`

The desired maximum duration in seconds of the trip that the responder defines. If no requirement is imposed, nil.

---

maximumWait: `Integer` */* nonnegative */* `|Nil;`

The number of seconds after which the `go` or `send` operation fails if the trip the responder defines isn't completed. If no requirement is imposed, nil.

Attributes (other)

notes: `Object|Nil;`

The agent's notes about the trip that the responder defines.

---

wayOut: `Way|Nil;`

The way from the origin of the trip that the responder defines to the destination. If no requirement is imposed, nil.

The `go` or `send` operation sets this attribute to nil.

Ticket Stub

Object
- **Ticket Stub**

Class

```
TicketStub: interface =
  (
    public
      see initialize, isConstrained, notes, wayBack
  );
```

An object that documents a trip, after the fact, from the viewpoint of the agent that took the trip. A ticket stub is derived from a ticket.

Constructor

initialize: `op (wayBack: Way|Nil; notes: Object|Nil);`

Sets the responder's attributes according to the following table.

Attribute	*Argument*	*Default*
wayBack	Yes	nil
notes	Yes	nil
isConstrained	–	*false*

Attributes

isConstrained: `Boolean;`

True if the trip the responder documents succeeded but resulted in a local or regional permit more constraining than the one the agent requested.

notes: `Object|Nil;`

The agent's notes about the trip the responder documents.

wayBack: `Way|Nil;`

A way by which the agent that made the trip that the responder documents can return to the agent's origin from the agent's actual destination. If the engine provides no way back even though such a way may exist, nil.

Time

Object
- **Time** (Ordered and Protected)

Class

```
Time: interface (Object, Ordered, Protected) =
   (
      public
         see adjust, asCalendarTime, interval, localize
   );
```

An object that identifies a date and time of day to the precision of 1 second or finer using UTC .

Note. A time identifies neither a time zone nor to what extent DST is in effect.

Specializations.

- The constructor makes the responder the current time.
- The `copy` operation regards as the responder's only property the integer that is the arithmetic difference between the responder (the minuend) and a fixed time (the subtrahend).
- The `order` operation relates two times as it does their properties.

Operations

`adjust: op (seconds: Integer) Time;`

If the `seconds` argument is negative, returns a time that is *N* seconds before the responder. If the argument is positive, returns a time that is *N* seconds after the responder. In either case, *N* is the argument's absolute value.

`interval: op (subtrahend: Time) Integer;`

Returns the arithmetic difference in seconds between the responder, the minuend; and the `subtrahend` argument, the subtrahend.

`localize: op () CalendarTime;`

Returns a normalized calendar time that denotes the same second in time as the responder and that reflects the permanent and seasonal offsets of the place that the current process occupies.

Conversions

asCalendarTime: `op () CalendarTime;`

Returns a normalized calendar time that denotes the same second in time as the responder and that reflects permanent and seasonal offsets of 0.

Trip Exception

Object
• Exception
• • **Trip Exception**

Class

```
TripException: abstract interface (Exception) =
(
   public
      see initialize, ticketStub
);
```

An exception thrown when a trip fails.

Constructor

```
initialize: op (ticketStub: TicketStub|Nil);
```

Sets the responder's attributes according to the following table.

Attribute	*Argument*	*Default*
ticketStub	Yes	nil

Attributes

```
ticketStub: sealed readonly TicketStub|Nil;
```

The ticket stub for the unsuccessful trip that the responder documents. If the responder is used in a context other than that of an unsuccessful trip, nil.

Subclasses

```
DestinationUnavailable: interface (TripException) = ();
```

A place is temporarily unreachable.

```
DestinationUnknown: interface (TripException) = ();
```

A place can't be identified and thus is permanently unreachable.

```
OccupancyDenied: interface (TripException) = ();
```

A place denies the process entry.

ReservationUnavailable: `interface (TripException) = ();`

A reservable means can't be reserved.

TicketExpired: `interface (TripException) = ();`

A place is unreachable in the maximum time allowed.

WayUnavailable: `interface (TripException) = ();`

A place lacks the required way out.

Uncopied

Uncopied

Class

```
Uncopied: mixin interface = ();
```

An object that can't be copied.

Specializations.

- The `copy` operation returns a reference to the responder, rather than create a copy. If the supplied reference is protected, so is the returned reference.
- The `encode` operation throws Encoding Unavailable.

Unmoved

Unmoved

Class

```
Unmoved: mixin interface = ();
```

An object that can’t be moved from place to place. If owned by an agent or its clone, an unmoved object is destroyed when the agent or clone moves.

Verified

Verified

Class

```
Verified: mixin interface =
   (
      public
         see verify
   );
```

An object that can become internally inconsistent in ways that depend upon the object's other member classes.

Operations

```
verify: abstract op () Boolean;
```

Returns *true* if the responder is consistent.

Way

Object
- **Way**

Class

```
Way: interface =
   (
      public
         see authenticator, entity, initialize, means
   );
```

An object that identifies and provides access to an entity, the way's *subject*. The subject may be, but needn't be an object itself.

Constructor

```
initialize: op (
   entity: Telename|Nil;
   means: Means|Nil;
   authenticator: Authenticator|Nil);
```

Sets the responder's attributes according to the following table.

Attribute	*Argument*	*Default*
entity	Yes	nil
means	Yes	nil
authenticator	Yes	nil

Attributes

```
authenticator: Authenticator|Nil;
```

How to authenticate oneself to the responder's subject. If no requirement is imposed, nil.

```
entity: Telename|Nil;
```

A telename of the responder's subject. If no requirement is imposed, nil.

```
means: Means|Nil;
```

How to reach the responder's subject. If no requirement is imposed, nil.

Appendix: Safety Limitations

This appendix lists the known safety limitations of this version of the language and its predefined classes.

Masquerade

- This vresion doesn't authenticate a class's name, authenticate, or identify a class's author. By including in its `publicPackages` attribute a class falsified in either respect, a place can alter the behavior of any occupants, of the place and its subplaces, that employ that class.
- The current object can exercise the privileges the language grants to either the current sponsor or the current owner. In general, the current object cannot request a feature of another object without allowing the latter, while the current object, to exercise those same privileges.

Leakage or loss of data

- The `encode` operation exposes the responder's properties to examination by including representations of them in the octet string it returns.
- One object cannot request an operation of another without giving the latter access to the information represented by the global variables. Such access allows the operation to have possibly unwanted side effects.
- By exercising its ownership privileges (see "Object ownership"), a process can modify objects it owns—and objects that other processes own that refer to those objects—in ways that adversely affect other processes.

  Note One can program processes to avoid or cope with the conditions above.

Denial of service

- A ticket's `destinationPermit` attribute is a request not a demand. Any place that satisfies an agent's ticket can let the agent enter without granting the capabilities the ticket requests (see the `isConstrained` attribute of a ticket stub). The destination region can do the same. The place or region can reduce the agent's capabilities thereafter. This puts the agent at risk, while in that place or region, of exhausting its permit.
- A region (rather than the language) assigns particular numbers of teleclicks to the expenditure of particular resources in particular amounts. A region's assignment of unexpectedly large numbers puts an agent at risk, while in that region, of exhausting its permit.
- A ticket cannot express, except by means of its `destinationAddress` attribute, an agent's possible unwillingness to leave the current region.
- One agent can force another to expend resources (however minimal) by trying to meet with it, thereby making it perform the `meeting` operation. In a similar way, an agent can compel a place to expend resources by

trying to enter it, thereby making it perform the `entering` operation. In this way the agent or place is put at risk of exhausting its permit.

- A subclass of Agent or Place can control the native permit and the requested local permit of a newly constructed agent or place. Even if the subclass's author represents that certain arguments of the constructor are used for those purposes, the subclass's method for the constructor is free to disregard those arguments when it escalates the constructor.
- A subclass of Meeting Place, which can define its own method for the (unsealed) `meet` operation, need not arrange meetings as the language describes. For example, it need not request the `meeting` operation of a petitionee or update the petitioner's or petitionee's `contacts` attribute.
- An agent engaged in a meeting can exit the place it occupies without ending the meeting. In this situation, the engine does not signal a part event. Hence the agent's acquaintances are not notified of its departure.
- The current object can signal a process event (and thus an exit event or a part event). Although the defined semantics of such events imply that only the engine can signal them, there is no such restriction.

Other limitations

- A place's method for the `entering` operation controls entry to that place but does not control entry to its subplaces. Each subplace's method for the `entering` operation independently controls entry to that subplace.
- Neither an agent constructed by an agent that is a possible duplicate nor a clone of an agent that is a possible duplicate is itself marked as a possible duplicate (see the `isPossibleDuplicate` attribute).

Index

This index encompasses the terms and identifiers that the manual defines. In general, only the defining occurrence of each term or identifier is indexed.

A

abstract 20
abstract program 57
acquainted 71
activations 72
actual arguments 8
actual class 47
actual destination 76
actual parameters 19
add 116, 136
address 175
AddressInvalid 183
adjust 203
Advertised places 65
after 157
age 166, 170
agent 67, 93
agentClass 172
agentName 172
agents 1
allowance 79
alphabetic characters 59
anchor 20
and 98
append 136
ArgumentInvalid 125
ArgumentMissing 125
arguments 8
asBitString 156
asCalendarTime 204
asCharacter 148
ascii characters 60
asIdentifier 197
asInteger 104, 148, 154, 156, 197
asList 111
asOctet 148
asOctetString 97, 148, 197
asReal 149, 197
asserted class 47
assigned teleaddress 65
assigned telename 73
assignments 170
asString 129, 149, 156
asTime 101
attribute 10
AttributeReadOnly 125
authenticator 69, 95, 210
authenticity 167
authority 1, 64, 73, 198, 199

B

base class 12
basic instance 23
before 157
binary telescript 62
Bit 96
BitString 97
block 9
Boolean 98
bottom 194
brand 72
break 57
byCopy 12
byOwner 12
byProtectedRef 12
byRef 12
byUnprotectedRef 12

C

calendar time 87
CalendarTime 99
canCharge 167
canCreate 167
canDeny 167
canGo 167
canGrant 167
canonical order 22
canRestart 167
canSend 168
cascade 28
Cased 102
catch 8
catchphrase 43
ceiling 148
Character 103
character telescript 62
charge 171
charges 167, 170
class 19, 105, 151
class family 19
class feature 19
class graph 22
class method 19
class search algorithm 13
class specifier 12
ClassAbstract 133
classDigest 108
ClassException 107
ClassName 108
ClassUnavailable 125
clear 110
clearEvents 122
clone 69
closure 14
Collection 109
CollectionException 112
comment 58
compare 113, 120, 191
Compared 113
compatible 12
compatibles 159
concrete 20
condition 77, 189
conditional use 77
conditions 190
ConditionUnavailable 183
ConditionUndefined 183
connection identifier 70
connectionID 127
constraint 11
constructed 23
constructor 23
contacts 180
control characters 60
controlled block 41
ConversionUnavailable 133
copy 15, 152
copy-equal 15

current 132
current class 25
current method 25
current object 25
current operation 25
current owner 25
current permit 82
current process 25
current sponsor 25
current stack 25

D

day 99
dayOfWeek 100
dayOfYear 100
DeathEvent 114
decimal digit characters 60
decode 155
decodes 4
decreased 81
default actual parameters 19
default parameters 19
defines 19, 20
derived class 19
desiredPriority 170
desiredWait 201
destination 67
destinationAddress 201
destinationClass 201
destinationName 201
destinationPermit 201
DestinationUnavailable 205
DestinationUnknown 205
destroy 16
Dictionary 115
difference 147, 192
disable 76
disableEvents 122
divide 147
DivisionByZero 178
drop 116, 136
dst 87, 100, 119
duplicate 67

E

effective package 74
electronic marketplace 1
empty 109
enableEvents 122
enables 76
encode 153
encodes 4
EncodingInvalid 133
EncodingUnavailable 133
engine 64
engine place 64
EnginePlace 118
entering 65, 176
entity 210
Equal 120, 157
escalated 23
EscalationInvalid 125
escape sequence 58
evaluated 28
event 75, 121
event selector 75
EventProcess 122
examine 14, 110
exception 8, 114, 124
exclude 110
exclusive use 77
executed 9, 28, 57
executes 43
ExecutionException 125
exhausts 82
existing connection means 69
ExistingConnectionMeans 127
ExitEvent 128
exiting 66
expires 67, 70
explicitly 9
expression 28
extent 167

F

fails 8, 9
false 98
features 19
FeatureUnavailable 125
find 117, 138, 161, 164
findClass 161
findPackage 161
fixed in number 8
flavor 20
floor 148
formal argument 36
formal parameters 19
freeze 17, 162

G

get 10, 11, 117, 138, 161
getEvent 123
getPackage 161
gets 10
getter 10
global variable 29
go 93

H

handles 43
host 73
hour 100

I

identifier 5, 129
identifiers 28
identity 199
immediate implementation subclass 20
immediate implementation superclass 20
immediate interface subclass 20
immediate interface superclass 20
immediate subplace 64
immediate superplace 64
implementation 9, 11, 19
implementation member 20
implementation member classes 20
implicitly 9
in force 32, 33
include 110
increased 81
inherited implementations 20
inherited interfaces 19
inherits 19, 20
initialize 95, 97, 99, 105, 108, 109, 114, 115, 119, 121, 127, 128, 145, 150, 155, 158, 160, 163, 164, 166, 169, 172, 175, 177, 180, 188, 189, 196, 198, 199, 200, 202, 205, 210
instance feature 19
instance method 19
instance property 19
instances 19
Integer 130
intended destinations 76
interface 8, 10, 19
interface member 20

interface member classes 20
InternalException 126
intersection 81, 168, 193
interval 203
isAlphabetic 103
isASCII 103
isConstrained 202
isDecimalDigit 103
isDone 132
isEqual 152
isFrozen 150
isInstance 151
isInstanceByName 151
isLocked 151
isLower 102
isMember 151
isMemberByName 152
isNilOK 12
isolate 17, 153
isOwned 151
isPossibleDuplicate 180
isProtected 151
isPunctuation 103
isReserved 188
isSame 152
isSpace 104
isSubclass 105
isSubclassByName 105
isSubclassOK 12
isUpper 102
items 109, 132
iterator 110, 132

K

KernelException 133
KeyInvalid 112
keys 115

L

language 3
language manual 6
length 109
lightweight primitive 12
List 135
live 181
local permit 80
local variables 10
localize 203
localPermit 170
location 198
lock 17, 152
lose contact 77
lowercase characters 60

M

magnitude 148
makeLower 102
makeUpper 102
MarkMissing 133
match 109, 113, 115, 135, 158, 192, 194
matches 83
maximum 157
maximumWait 173, 201
means 69, 139, 210
meet 2, 70, 142
meeting 70, 140
MeetingAgent 140
MeetingDenied 141
MeetingDuplicated 141
MeetingException 141
MeetingPlace 142
metacharacters 83
method 9
minimum 157
minute 100
MiscellaneousException 144
mix-in 20
modify 14
module 30
modulus 130
month 100
multiply 147

N

name 105, 145
Named 145
native implementation 20
native interface 19
native permit 79
nativePermit 171
negate 148
new 106
next 132
Nil 146
noncharacters 60
nonspacing mark characters 60
normal priority 73
normalize 100
normalized 88
not 98
notes 201, 202
null 28
Number 147

O

object 8, 150
object program 62
ObjectFrozen 133
ObjectUninitialized 133
ObjectUnowned 134
OccupancyDenied 206
occupied 65
occupy 64
Octet 154
OctetString 155
offers 74
one 96
operation 8
or 98
order 157
Ordered 157
origin 67
out method 9
owned 16
owner 151

P

package 74, 158
PackageProcess 160
PackageUnavailable 183
parameters 19
part 70, 143
partAll 143
PartEvent 163
passage 12
pattern 83, 164
PatternInvalid 144
peers 73
performed 9
permanent offset 87
permanent permits 79
permit 78, 166, 170
PermitExhausted 126
PermitProcess 169
permits 1
PermitViolated 183

petition 3, 70, 172
petitionee 70
petitioner 70
PetitionExpired 141
place 64, 174
place hierarchy 64
PlaceNotCurrent 183
places 1
pop 194
PositionInvalid 112
positions 135
predefined class 19
predefined method 9
Primitive 177
PrimitiveException 178
priority 73, 167, 170
private operation 8
private use characters 60
privately 74
privatePackages 160
process 72, 179
processes 72
ProcessEvent 182
ProcessException 183
ProcessNotControlled 183
ProcessNotCurrent 184
produces 132
ProgrammingException 185
propagates 8
properties 11
PropertyUndefined 126
protect 152
Protected 186
protected object 15
protected reference 14
public operation 8
publicly 74
publicPackages 160
punctuation characters 60
push 194
pushItems 194

Q

quotient 131

R

read only 10
Real 187
receives 76
record 66, 71, 128, 163
reference 14
ReferenceProtected 126
references 3
ReferenceVoid 126
referent 14
region 64
regional permit 80
regionalData 181
regionalPermit 171
rekey 116
reposition 136
request 8, 10
requester 8, 10
reservable means 69
ReservableMeans 188
reservation 69
reservation interval 69
ReservationUnavailable 206
reserve 188
reserved 69
reserved word 59
resource 77, 189
responder 8, 10, 28
ResponderMissing 126
ResponderNil 126
result 8, 28
ResultInvalid 126
ResultMissing 126
return 8
roll 195
round 148
Routing advice 65
routingAdvice 198

S

Same 191
satisfied 68, 70
satisfies 83
satisfy 12
sealed 20
seasonal offset 87
second 100
securityRegime 95
send 94
sends 76
set 10, 11, 116, 137, 192
sets 10
setter 10
shall 27
shallowCopy 110
shared use 77
signal 75
signalEvent 123
signature 34, 35
size 16, 151
source 76, 121
source program 57
space characters 60
special characters 60
specifies 19, 20
splice 137
split 165
sponsored operation 8
Stack 194
StackDepleted 112
statement 28
String 196
subject 78, 83, 108, 166, 210
sublist 135
subplaces 64
substitute 165
substring 83, 197
subtype 12
succeeds 8, 9
sum 147
SuperclassesInvalid 107
superplace 64
supertype 12
swap 195

T

teleaddress 1, 65, 198
teleclicks 79
telename 1, 73, 199
Telescript 91
Telescript engine 3
telescript escape 56
Telescript language 2
Telescript model 1
telesphere 4, 64
temporary permit 80
terminates 73
text 29, 83
thaw 18, 162

this version 4, 10, 11, 12, 21, 22, 29, 30, 32, 33, 34, 36, 53, 211
throws 8
ticket 2, 67, 200
ticket stub 68
TicketExpired 206
TicketStub 202, 205
time 121, 203
top 194
transpose 117, 137
trip 67
TripException 205
true 98
type 12

U

unconditional use 77
Uncopied 207
Unicode specification 6
uniformly owned 17
union 193
unlockedCopy 152
Unmoved 208
unordered 157
unprotected reference 14
unsponsored operation 8
uppercase characters 60
user-defined class 19
user-defined method 9
uses 77
utc 87

V

value 9, 28, 43, 57
values 115
variable declaration 9
variable declaration segment 9
VariableUndefined 126
varying in number 8
Verified 209
verify 209
violates 82
virtual places 64
visited 1
voided reference 14

W

wait 181
way 69, 210
wayBack 202
wayOut 201
WayUnavailable 206
who 181

Y

year 100

Z

zero 96, 154
zone 100, 119