Draft American National Standard for Information Systems - Programming Languages - Smalltalk

Notice

This is a draft proposed American National Standard. As such, this is not a completed standard. The Technical Committee may modify this document as a result of comments received during public review and its approval as a standard.

Copyright

Table of Contents

Forward

Smalltalk is designed to be a "single paradigm language with very simple semantics and syntax for specifying elements of a system and for describing system dynamics." The principle is explained by the designers of the original Smalltalk-80 language.

There is a continuing growth of interest in the language. Its use has spread beyond the education and research community to the commercial applications in recent years. Data from many sources (including polls in conferences and reports from independent consultants) indicate the growing popularity of Smalltalk as an object-oriented programming language.

There are currently at least five vendors of Smalltalk implementations. Although the actual number of Smalltalk users is unknown, we believe it to be high. (It has been estimated that Digitalk Inc., alone, had sold over 100,000 Smalltalk/V licenses by 1993.)

The growth, spread and potential of Smalltalk led to a need for a standard that will protect the users' interest in compatibility and portability. The J20 technical committee was formed in the summer of 1993 to develop the ANSI Smalltalk standard. Many people and organizations in and outside of the committee have contributed to the document. The following is the list of the formal committee members.

Yen-Ping Shan, Chairman
Glenn Krasner, Vice-Chairman
Bruce Schuchardt, Project Editor
Rick DeNatale, Secretary

Organization Represented	*Name of Representative*
Andersen Consulting	Philippe Monnet
The Gallery Group, Inc.	Tony Click
GemStone Systems, Inc.	Bruce Schuchardt
	Patrick Logan
IBM	Yen-Ping Shan
	Rick DeNatale
	Brian Barry
	David G. Thomson
Intuitive Systems, Ltd.	Blair McGlashan
McKesson Corp.	Douglas Surber
ParcPlace-Digitalk	Glenn Krasner
Quasar Knowledge Systems, Inc.	David Simmons
	Andrew Demkin
Texas Instruments, Inc.	Andy Hoffman
	Mary Fontana

In addition, the following individuals have made significant contributions to the development of this standard:

Allen Wirfs-Brock
David N. Smith
Brian Wilkerson
Aaron Underwood
Thom Boyer
Daniel Lanovaz
Ken Huff
Juanita Ewing
Pat Caudill
Fred Chan
Mike Kahl
Bruce Conrad
Jim Fulton

1. Goals and Scope

The goal of the J20 Committee was to produce a written standard for the Smalltalk language such that:

1. working only from the standard, a conforming implementation can be produced,
2. Smalltalk programs which conform to the standard will have the same execution semantics on any conforming implementation, and
3. the standard shall be sufficiently complete to allow useful Smalltalk programs to be constructed.

The standard does not specify the full range of class libraries one would expect to find in a Smalltalk implementation. For example, it specifies neither user-interface nor database frameworks. It does provide facilities to create such classes that will work on compliant implementations having identical external facilities to support them.

Subject to the preceding points, the standard should:

1. constrain the nature of current and future implementations as little as possible, and
2. minimize impact on legacy code and implementations.

Although it was not the intent of the committee to produce a conformance tool or conformance test suite, the ability to define such conformance measures should be considered as a test of whether the standard is adequately unambiguous.

The following chapters specify the Smalltalk language in a way that is consistent with these goals. Chapter 2 specifies the terminology of conformance used in the standard. Chapter 3 specifies the language syntax and semantics. Chapter 4 specifies a standard interchange format for Smalltalk programs. Chapter 5 specifies the standard class library. Chapter 6 is a glossary of terms used in the document. This is followed by an index of the protocols found in Chapter 5 and a list of references that, while not part of the standard, are referred to in the text of the standard.

Most current Smalltalk implementations provide syntax and semantics only for Smalltalk methods. In particular, they do not provide an explicit definition of program construction, for example class creation and global creation and initialization. These program constructs, rather, are provided via some combination of programming tools and imperative operations, i.e. the evaluation of expressions in the language.

The Committee decided that neither tools-based definitions nor imperative-based definitions of these important program constructs were adequate for a language standard. As a result, Chapter 3 not only provides syntax and semantics for Smalltalk methods, but also gives an explicit, declarative syntax and semantics for all constructs in a Smalltalk program.

Chapter 4 gives a syntax for the format of files to be used for the interchange of Smalltalk programs among conforming implementations. The Standard is not defined in terms of file contents, but this file format syntax is intended to allow conforming programs to be moved between conforming implementions.

The Object Library specification in Chapter 5 has scope that meets the Committee's goals, and is implemented in a way that allows for specificity and allows for a significant amount of implementor latitude.

The scope of the Library is roughly an intersection of existing Smalltalk products' libraries. That is, it includes numbers, data structures (collections), basic objects (nil, Booleans, etc.), blocks, exceptions, and files. The intent is that the functionality specified would be both complete and adequate to use for interesting applications. The standard attempts to minimize these definitions

within the constraints of completeness and usability, so that implementors are not burdened with providing significantly more functionality than needed and so that the size of the base library could be kept relatively small.

The standard does not attempt to specify areas where current implementations differ in significant ways. In particular, as the goal statement implies, we did not include graphics, user interface, or database accessing objects in the library. Future revisions of this Standard may include a wider scope, especially if implementations converge.

Traditionally Smalltalk class libraries has been specified via their implementation, providing the definition of a particular set of classes, including their state (instance variables) and method implementations. This approach has major drawbacks to use as a library specification; it constrains implementors to using exactly the implementation specified, and it can lead to less verifiable specifications.

Rather than take this approach, we specify the Library in terms of the external behavior of the library objects. Implementors can take advantage of various implementation techniques as long as they deliver the specified external behavior. And this behavior must be rigorously specified.

The main drawback of this approach, in particular with respect to the implementation-based approach, is that the standard cannot specify the behavior of programs that subclass classes that implement the Standard Library. This is because, the behavior of such a subclass would be dependent upon implementation details of its superclass that are beyond the scope of this standard. Implementations are free to have instance variables and internal methods in their implementations of these classes and those variables and methods cannot be guaranteed not to conflict with compliant programs' instance variables and methods. For this reason, compliant programs cannot subclass most classes that implement the Standard Library. The standard does specify a limited set of classes, including most notably Object, may be subclassed by compliant programs. Implementation are required to implement these classes in a manner that will not conflict with the definition of subclasses.

The standard uses a particular technique for specifying the external behavior of objects. This technique is based on a protocol, or type system. The protocol-based system is a lattice (i.e. multiply inherited) based system of collections of fully-specified methods that follow strict conformance rules (which, by the way, is possible with protocol/type systems but is not feasible in implementation-based systems).

A protocol will specify the external behavior of those objects in the Library that the Standard defines. In addition, a relatively small number of named Globals, objects whose behavior is defined by a protocol, are specified. From these Globals, from the syntax-defined objects (e.g., program literals), and from the objects returned from messages sent to these objects, are produced the full set of objects defined in the Standard.

Note that the protocol mechanism is only of use to the specification, it is not a part of the Smalltalk language. The standard does not require implementations to implement a protocol mechanism. Implementations of the Standard only must provide object that conform to the protocol's specifications. These objects may be instances or they may be classes; there is no requirement that even the "class-like" protocols need to be implemented with classes. The standard does not require that each protocol isbe implemented with separate objects--there may well be implementations where single objects conform to multiple protocols. As long as the external behavior of the objects is what is specified, then the implementation is conforming and programs that use these objects should be conforming.

2. Conforming Implementations and Programs

A Smalltalk implementation conforms to this standard if it implements all defined features as specified in the standard.

In places the standard defers particulars of a feature to an implementation while still requiring that the feature be implemented. Such a feature is denoted *implementation defined* and a conforming implementation must document it.

If a feature is denoted *undefined* a conforming implementation may accept a program using the feature, but must document that it does so. A conforming implementation is not required to accept an undefined feature. A program that is dependent upon the use of an *undefined* feature does not conform to the standard.

There are also places where the standard explicitly denotes something as *unspecified.* Most notably, the protocol section allows an *unspecified* return value. The specifics of such features are not defined by the standard although some reasonable behavior is required. Conforming implementation must support the feature but need not documented the details of the implementation.

A Smalltalk program conforms to the standard if it only makes use of features defined in the standard. It may use *implementation-defined* features or *unspecified* featuresand still be considered a conforming program, though doing so may limit the program's portability.

If a feature is denoted *erroneous* a conforming implementation must reject a program using the feature.

The following table shows how a conforming implementation treats language features.

	Implementation-Defined	Unspecified	Undefined	Erroneous
Must Accept	√	√		
Must Reject				√
May Vary	√	√	√	
Must Document	√		√ **if accepted**	

3. The Smalltalk Language

A Smalltalk program is a description of a dynamic computational process. The Smalltalk programming language is a notation for defining such programs. This definition of the Smalltalk language consist of two parts. The first part defines the abstract computational model for Smalltalk programs. It defines the environment within which a Smalltalk program executes and defines the entities that exist within that environment. The second part defines the notation used to specify Smalltalk programs. It defines the syntax and semantics of the language. Taken together the two parts are intended to define the semantics of a Smalltalk program, but avoid requiring any specific implementation techniques.

The Smalltalk language defined in this chapter is an uniformly object-oriented programming language. It is uniform, in the sense that all data manipulated by a Smalltalk program is represented as objects. The language is a descendent of Smalltalk-80 [Gold83,Gold89]. The primary difference between ANSI Smalltalk and Smalltalk-80 is that ANSI Smalltalk provides for fully declarative specification of Smalltalk programs. In addition, implementation dependencies and biases have been eliminated from the language.

Rationale

Smalltalk program construction has traditionally been performed in the context of a "virtual image" [Goldberg93]. A virtual image consists of a set of objects. These objects include not only those that define a class library that is intended to be used and extended by application programs, but also objects that implement the interactive Smalltalk programming environment itself. In such an environment, a Smalltalk application program is constructed by directly or indirectly executing imperative Smalltalk expressions that extend and modify the objects within the virtual image to include the classes and variables necessary to implement the functionality of the program. Smalltalk has not included the concept of a program as a distinct entity. In practice, a Smalltalk program is the state of a virtual image when work is declared completed. The image contains the objects that are the implementations of classes, global variables and pools, but not the imperative expressions that created them. Therefore, to transfer a program to another virtual image, it is necessary to synthesize and externalize expressions that will recreate the program elements. However, the types of some program elements may not be readily discernible by examining their implementation artifacts. For example, in some implementations it is not possible to distinguish a pool from a global variable whose current value is a dictionary with strings for keys. More generally, it is not possible to synthesize the original initialization expressions for global variables. It is only possible to produce expressions that reproduce their current values.

Lack of a program definition in traditional Smalltalk environments leads to an undue reliance on the virtual image. Images can become obsolete or broken. Because the program is encoded in the image, the program is in danger of becoming inaccessible if the image becomes outmoded or corrupt.

Smalltalk's imperative program construction model also requires that the same virtual image be used both for program creation and program delivery. This makes it very difficult to support situations where the development must be performed in a computing environment that is different from the target execution environment.

Because of the issues identified above this standard has chosen to use a declarative model to define Smalltalk programs. This requires the introduction of additional declarative abstractions to the language for program elements that previously had only been defined in terms of implementation artifacts. All elements of a Smalltalk program are described existentially at a level of abstraction that does not overly constrain implementations of the language. The meaning of such a Smalltalk program should be understandable solely from the definition of the program without actually executing a program construction processor or making use of a pre-initialized execution environment.

The use of a declarative specification model has little direct impact upon Smalltalk programmers. Even though Smalltalk has traditionally been implemented using an imperative program description model, the perception of most Smalltalk programmers is of a declarative model. This is because Smalltalk programmers typically create and edit programs using a browser that presents the classes that make up the program in a declarative style.

With the declarative language model the use of reflection is not required (although it is permitted) in order to define a Smalltalk program. The following are a few other implementation assumptions made about the traditional execution environment for Smalltalk programs that are eliminated using this declarative model:

- A system dictionary exists.
- All classes, globals, and pools are in this system dictionary.
- Pools are realized using dictionaries.
- Global and pool variables are represented as associations.

- Each class has an associated metaclass.
- Methods are objects.

It is possible to create an implementation that conforms to this standard that contradicts none of these assumptions. However, it is equally possible to create a conforming implementation where none of the assumptions is true.

3.1 Computational Model of Smalltalk Execution

A Smalltalk program is a means for describing a dynamic computational process. This section defines the entities that exist in the computational environment of Smalltalk programs.

A *variable* is a computational entity that stores a single reference (the *value* of the variable) to an object.

A *message* is a request to perform a designated computation. An *object* is a computational entity that is capable of responding to a well defined set of messages. An object may also encapsulate some (possibly mutable) state.

An object responds to a message by executing a *method*. Each method is identified by an associated *method selector*. A *behavior* is the set of methods used by an object to respond to messages.

A method consists of a sequence of expressions. Program execution proceeds by sequentially evaluating the expressions in one of more methods. There are three types of expressions: assignments, message sends, and returns.

An *assignment* changes the value of a variable.

A *message send* causes execution of the currently active method to be temporarily suspended and for program execution to continue starting with the first expression of another method. A message send directs a message to an object. The object is called the *receiver* of the message. A *message* consists of a method selector and a set of arguments. Each argument is a reference to an object. When an object receives a message, the method selector of the message is used to select the method from the object's behavior that corresponds to the selector. The method becomes the new locus of execution. Special processing takes place if the receiver's behavior does not include a method corresponding to the message's method selector.

A *return* terminates execution of a method and causes execution to resume within the method that executed the message send that activated the method containing the return. Execution continues immediately following the message send expression. The return provides a value (an object reference) that becomes the value of the message send.

Within the text that defines a Smalltalk program, identifiers, called *variable names,* are used to refer to variables. A variable name is bound to a variable over some extent of the program definition. The extent within a program of such a binding is called the *scope* of the variable. The only operations a program may perform upon a variable are to access its current value or to assign it a new value.

The encapsulated state of an object consists of a (possibly empty) set of variables. Such variables are called *instance variables*. Normally each variable is bound to an associated instance variable name. The state of an object may also include a set of unnamed instance variables that are accessed via numeric indices. In addition, the state of an object may be represented in an implementation dependent manner. Other than instance variables, all variables are discrete execution entities that exist independently of any objects. Variables that are not instance variables are called *discrete variables*. A discrete variable whose scope is the entire program is a *global variable*. Other types of discrete variables are *pool variables*, *class variables*, and *temporary variables.*

The objects that exist during program execution consist of both *statically created objects* and *dynamically created objects*. A statically created object is an individual object that is explicitly

defined by the text of a Smalltalk program. Typically these are either *literals* or *class objects*. Some statically created objects are bound to an object name within some scope. Such objects are called *named objects*. The most commonly occurring named objects are class objects.

Dynamically created objects are not individually defined by the program, instead they are dynamically created as a side effect of the execution of a method. Dynamically created objects do not have names. They are typically referenced as the value of a variable.

Smalltalk does not provide an explicit mechanism for destroying objects. Conceptually, once an object is created it continues to exist for the duration of the program's execution. In practice, many objects are used for only a portion of a program's execution and then logically ignored. Smalltalk implementations use *garbage collection* techniques to automatically detect and reclaim the resources associated with objects which are no longer of use to the program. During program execution each object must continue to exist, preserving its state, for as long as it is possible to execute any statement that may reference a variable having that object as its value. The garbage collector is free to reclaim the resources associated with any object when it can prove that continued execution of the program will never reference that object.

Immediately prior to the execution of a Smalltalk program all statically created objects are in their initial state as defined by the Smalltalk program and the values of all discrete variables are undefined. Execution proceeds by sequentially executing each *initializer* in the order specified by the program definition. If a program accesses any variable that has not been explicitly initialized either by an initializer or by an assignment statement its value will be the object named `nil`.

Rationale

The vast majority of Smalltalk application programs do not utilize the reflective capabilities available in traditional Smalltalk implementations. For this reason, we view such reflective capabilities as artifacts primarily used in the implementation of incremental program facilities and do not mandate their presence in all Smalltalk implementations. Given this view, the standard execution model only needs to define the entities that are part of a programmer's view of a running Smalltalk program. These are variables and objects with state and behavior. Implementation artifacts such as compiled methods, method dictionaries, or associations representing variables are excluded.

Class objects have no special significance other than having names and having behaviors and state distinct from that of their associated instance objects. Unlike classic Smalltalk definitions [Goldberg83], they are not defined as being the containers or implementers of their instances' behavior. The techniques used to implement the behavior of objects is left to the implementers. Finally, because classes are not specified as the implementers of behavior, metaclasses are not needed to provide the behavior of class objects.

3.2 The Syntax of Smalltalk Programs

The Smalltalk programming language provides the notation for defining Smalltalk programs. The previous section defined the computational elements of such a program. This section defines the notation.

Traditionally, Smalltalk has not had an explicit notation for describing an entire program. An explicit syntax definition has been used for certain program elements, such as methods, but the manner in which such elements were structured into a complete program was defined by the various implementations. This standard recognizes the need for a standard definition of the structure of Smalltalk programs but follows the Smalltalk tradition of not mandating a single universal linear textual representation for programs. It accomplishes this by specifying an *abstract syntax* for Smalltalk programs. The abstract syntax specifies all elements that make up a Smalltalk program and the manner in which such elements are logically composed of constituent elements. However, a single concrete representation of the abstract program syntax is not mandated. Implementations are free to define and support various concrete syntaxes that conform to the standard abstract syntax for Smalltalk programs. One such concrete syntax that all implementations are required to support is the "Smalltalk Interchange Format" that is defined in section 4 of this standard.

The notation for Smalltalk programs are defined by three inter-related specifications. The *lexical grammar* specifies the valid sequences of individual characters that make of the tokens of the Smalltalk language. These token serve as the "atoms" out of which Smalltalk program definitions are constructed. The *method grammar* specifies the concrete syntax for creating methods and

initializers from tokens. The *abstract program grammar* specifies the abstract syntax of a complete Smalltalk program.

The lexical grammar defines a concrete syntax for tokens and the method grammar and for methods and initializers. They define a single, specific language for defining such entities. The abstract program grammar does not define a concrete syntax. Instead, it specifies the logical structure of a Smalltalk program and identifies optional and required program elements.

A variant of Extended Backus-Naur Form (EBNF) is used to define each of the three grammars. The symbols that make up the productions of the grammar are either identifiers that name syntactic categories or literal tokens enclosed in single quotes. The names of syntactic categories of the program grammar are enclosed by double angle brackets (for example:<<progamElement>>). The names of syntactic categories of the method grammar are enclosed by single angle brackets (for example: <assignment>). Identifiers that are neither quoted or enclosed by angle brackets are names of syntactic categories of the lexical grammar (for example: identifier). Productions of the abstract program grammar may reference symbols of either the method grammar or lexical grammar. Productions of the method grammar may reference symbols of the token grammar.

Within productions, alternatives are separated by a vertical bar ('|'), and typically listed on separate lines. Optional symbols are enclosed in square brackets ('[' and ']'). Symbols may also be grouped using parenthesis. The plus sign ('+') following a symbol or group of symbols indicates one or more repetitions; an asterisk ('*') indicates zero or more repetitions. For example: the following rule:

```
['('] <statement>+ [')']
```

defines a sequence of one or more statements that are optionally enclosed in parentheses.

Within the concrete lexical and method grammar, the ordering elements of the elements of a product define a required syntactic ordering. Within the abstract program grammar, the ordering of the elements of a production do not define or imply any such syntactic ordering The productions of the abstract program syntax are only used to define the constituent elements of a program. Ordering and other structural attributes are a characteristic of concrete program syntax and except for the "Smalltalk Interchange Format" are outside of the scope of this standard. An optional symbol in an abstract program grammar production identifies a program element that is optionally present. Repeat symbols indicate that multiple instances of the program element may be present. The occurrence of a lexical grammar or method grammar symbol in a production of the abstract program grammar means that any concrete form of the abstract production must include the concrete language element produced by the symbol's production.

Abstract grammar identifiers that are in italics (for example, <<*flag*>>) are "atoms" that do not have further structure or contain any lexical or method grammar elements. Typically such identifiers correspond to optional attributes of program element definitions. Any concrete program syntax must provide a mechanism for specifying these attributes.

3.3 Smalltalk Abstract Program Grammar

3.3.1 Program Definition The definition of Smalltalk programs consists of a sequence of program element definitions. The program element definitions define all discrete variables, statically created objects and the behaviors of all objects that will take part in the computation. In addition, the program definition specifies the order of dynamic initialization for all program elements.

```
<<Smalltalk program>> ::= <<program element>>+ <<initialization ordering>>
<<program element>> ::=  <<class definition>> |
<<global definition>> |
<<pool  definition>> |
<<program initializer definition>>
```

The concrete syntax, representation, and character encoding of a <<Smalltalk program>> is

implementation defined. A concrete program representation may define syntax for grouping or structuring subsequences of <<program elements>> into individual storage units. Such units must be logically composable into a valid <<Smalltalk program>>.

The <<program element>> clause of a <<Smalltalk program>> logically includes the definitions of any standard or implementation program elements used by the program. An implementation may define a mechanism for the automatic inclusions of such definitions.

Some program elements define an identifier as a global name that uniquely identifies the program element. These global names exist in a single global name scope that is in the scope of every <<program element>> within the program. Each global name must be uniquely defined by a single <<program element>>. It is erroneous if two or more <<program element>>definitions use the same identifier as a global name. The ordering of the <<program element>>definitions should have no affect upon the visibility of global names. A <<program element>> may reference any global name regardless of whether the definition of the global name proceeds or follows the <<program element>>.

The <<*initialization ordering*>> defines the order in which the initialization of each individual <<program element>>. will occur. Program execution begins with the execution of the initializer of the first <<program element>> and proceeds by executing, in turn, the initializer of each subsequent <<program element>>. Execution of the program terminates after completion of the initializer of the last <<program element>>.

3.3.1.1 Name Scopes

Within a program definition identifiers, called *names*, are used to refer to various entities. Such a name is said to be *bound* to an entity. An occurrence of the name within the program definition is interpreted as a reference to the entity to which the name is bound. The association between a name and an entity is referred to as the *binding* of the name. A name may have different bindings at different points within a program definition. A *name scope* is a set of name bindings that are available to some portion of the Smalltalk program definition. A reference to a name at some point in a program definition is *resolved* to the specific binding of the name that exists in the scope that is available at that point. The binding of a name within a scope may be specified as an *error binding*. Any reference to a name which resolves to an error binding is erroneous.

A name scope may be defined as a composition of other, already defined, name scopes. Two name scopes are composed by specifying one of them as the *outer scope* and the other as the *inner scope*. The set of names in such a *composite scope* is the union of all the names in the outer scope and all of the names in the inner scope. The binding of each name is the binding of the name from the inner scope, if the name occurs in the inner scope. Otherwise, the binding of the name, is the binding of the name from the outer scope. If a binding for the same name appears in both the inner scope and the outer scope, the inner scope binding is said to *shadow* the outer scope binding. It is the inner scope binding that is available as part of the composite scope.

In this document, an algebraic notation is sometimes used to define composite scopes. The "+" operator is used to define a composite scope where the scope on the left of the "+" is the outer scope and the scope on the right of the "+" is the inner scope. Parenthesis are used to specify complex composite scopes. For example, the expression:

(outer + middle) + inner

describe a composite scope whose outer scope is itself a composite.

A <<Smalltalk program>> introduces a name scope, called the *global scope*, that is available to all parts of the program. The global scope is a composite scope whose outer scope is called the *extension scope* and whose inner scope is called the *global definition scope*. The global definition scope contains name that are explicitly defined by <<program element>> definitions. The extension scope exists so that Smalltalk implementations may provide predefined global name bindings. The mechanism, if any, for adding name bindings to the extension scope of a <<Smalltalk program>> is unspecified.

Global names introduced as part of the implementation of the standard class library or as part of an implementation's extended class library have the potential of interfering with the portability of user programs. A standard conforming program might not be portable to an implementation if that implementation defines a global that is the same name as one of the program's global.

Rationale

The extension scope exists so that any implementation specified predefined names may be defined a scope that is shadowed by all other scopes. This enhances program portability between implementations by ensuring that any coincidental explicit definition of a name will take precedence over any implementation specific binding of the same name.

Three ways to deal with the global name clashes were discussed:

1. Recognize and ignore it.
2. Require that any such global names used by the application use underscore prefixes. This is unlikely to be an acceptable solution as it would essential require all implementation provided globals that are not required by this standard to use such names even though they are likely to be referenced by user code.
3. Require that all implementation globals be defined in the extension scope. Any conflicting user program globals would shadow the implementation globals. This would generally solve the portability problem but its implementation would require significant changes to existing implementations.

A standard conforming implementation may use any of these approaches.

3.3.2 Class Definition

A *class definition* defines the behavior and encapsulated state of objects. In addition, a class definition introduces a named object binding within the global definition scope. This name is called a *class name* and the associated object is a *class object*. A class definition specifies the behavior and instance variable structure for both the statically created class object and any dynamically created instances of the class.

A class definition specifies two behaviors, the *instance behavior* and the *class behavior*. The instance behavior is the behavior of any dynamically created instances of the class. The class behavior is the behavior for the class object. Through the use of inheritance, the instance variable structure and behavior for both the class object and instance objects may be specified as a refinement of that specified by another class definition known as its *superclass*. Conversely, a class definition that inherits such structure or behavior is known as a *subclass* of its superclass.

A class definition may also define discrete variables called *class variables* whose scope is all methods (either class or instance methods) defined as part of the class definition or as part of the class definitions of any subclasses of the class definition. In addition a class definition may specify the importation of pools. Any pool variables defined in such pools are included in the scope of all methods defined as part of the class definition.

```
<<class definition>>::=
        <<class name>> [<<superclass name>>]
                [<<instance state>>]
                [<<class instance variable names>>]
                [<<class variable names>>]
                [<<imported pool names>>]
                [<<instance methods>>]
                [<<class methods>>]
                [<<class initializer>>]
<<class name>> ::= identifier
<<superclass name>> ::= identifier
<<instance state>> :=
        <<byte indexable>> |
         [<<object indexable>>]  <<instance variables names>>
<<instance variables names>> ::= identifier*
<<class instance variable names>> ::= identifier*
<<class variable names>> ::= identifier*
<<imported pool names>> ::= identifier*
<<instance methods>> ::= <method definition>*
```

```
<<class methods>> ::= <method definition>*
<<class initializer>> ::= <initializer definition>
```

The <<class name>> is the global name of the class object. This binding is contained in the global definition scope. It is erroneous if there is any other global definitions of this name within the program. The binding of the <<class name>> to the class object is a constant binding. It is erroneous for an identifier that resolves to such a binding to appear as the target of an assignment statement. It is erroneous if the <<class name>> is one of the reserved identifiers, "true", "false", "nil", "self" or "super". Class names whose initial character is an underscore are reserved for implementation use.

The <<superclass name>> identifies the class definition from which this definition inherits. It is erroneous if <<superclass name>> is not the <<class name>> of another <<class definition>> whose binding exists in the global scope of this program. If the <<superclass name>> is absent then this class has no inherited behavior. It is erroneous if the <<superclass name>> is the same as the <<class name>> or if <<superclass name>> is the name of a class that directly or indirectly specifies <<class name>> as its <<superclass name>>.

3.3.2.1 Instance State Specification

The <<instance state>> production defines the representation of the state that is encapsulated by objects that are instances of the class. The state consists either of variables that reference other objects or variables that store binary data. The number of state variables may either be the same for all instances or may vary between different instances.

The <<*byte indexable*>>and <<*object indexable*>> productions specify that the instances of this class encapsulate a variable number of unnamed instance variables. The individual unnamed instance variables are identified using numeric integer indices that range from 1 to the total number of unnamed instance variables. The actual number of unnamed instance variables associated with a particular instance of the class is determined at the time the instance is created. The meaning of the class definition is undefined if a class definition includes a <<*byte indexable*>> clause and any superclass definition includes an <<instance variable names>> or an <<object indexable>> clause. The meaning is undefined if a class definition includes an <<instance variable names>> or an <<object indexable>> clause and any superclass definition includes a <<*byte indexable*>> clause. If a class definition does not include an <<instance state>> clause then the representation of instances of the class is the same as the representation of the instances of its superclass. If a class definition does not include an <<instance state>> clause and the class definition does not specify a superclass then instances of the class have no instance state.

If the <<*object indexable*>> clause is present in the definition of a class or any of its superclasses the values of the unnamed instance variables are object references. When such an object is created the initial value of each unnamed variable is nil. An object whose state includes such variables is called an *indexable object*. An indexable object may also have named instance variables.

If the <<*byte indexable*>>clause is present in the definition of a class or any of its superclasses the values of the unnamed instance variables are restricted to be integers in the range 0 to 255. When such an object is created the initial value of each unnamed variable is 0. An object whose state includes such variables is called a *byte indexable object*.

The <<instance variable names>> production defines the names of instance variables of the objects that are instances of the class. The identifiers specified by <<instance variables names>> are called *instance variable names*. It is erroneous for the same identifier to occur more than once in the sequence of instance variable names. The meaning of the class definition is undefined if any of the instance variable names is the same identifier as an instance variable name or class variable name defined by any superclass. It is erroneous for an instance variable name to also appear as a class variable name. It is erroneous if an instance variable name is one of the reserved identifiers, "true", "false", "nil", "self" or "super". The *complete instance variable set* is the set consisting of the

union of the set of instance variable names of the <<class definition>> and the complete instance variable set of the class definition's superclass. If a <<superclass name>> is not specified in a class definition its complete instance variable set is simply the set of instance variable names of the <<class definition>>.

Rationale

The meaning of a class variable name, instance variable name, or class instance variable name that is the same as an inherited variable name is intentionally left undefined in order to permit implementations to extend this standard such that such a definition would shadow the inherited definition. While Smalltalk implementations have traditionally treated such shadowing as an error, shadowing is necessary in order to avoid the "fragile subclass" problem" It is anticipated that implementations that wish to address this problem will allow such shadowing.

The encapsulated state of an instance object for a class definition that includes an <<instance variable names>> clause consists of a fixed-size set of variables capable of referencing any object. The identifiers specified by <<instance variable names>> are called *instance variable names* and the associated variables are called *named instance variables*. The number of named instances variables encapsulated by an instance object is equal to the size of the complete instance variable set of the object's class definition. There is a one-to-one correspondence between the members of the complete instance variable set and the named instance variables of an instance object. When an instance of a class is created all of the named instance variables initially have the value of the reserved identifier nil. An object with named instance variables may also have unnamed indexable instance variables.

3.3.2.2 Class State Specification

The <<class instance variable names>> production defines the names of instance variables of the class object. The identifiers specified by <<class instance variables names>> are called *class instance variable names*. It is erroneous for the same identifier to occur more than once in the <<class instance variable names>> clause. The meaning of the class definition is undefined if any of the class instance variable names is the same identifier as a class instance variable name or a class variable name defined by any superclass. It is erroneous for a class instance variable name to also be a class variable name. It is erroneous if a class instance variable name is one of the reserved identifiers, "true", "false", "nil", "self" or "super". The *complete class instance variable set* is a set consisting of the union of the set of class instance variable names of the <<class definition>> and the complete class instance variable set of the class definition's superclass (or if a <<superclass name>> is not specified, a possibly empty set of implementation defined underscore-prefixed class instance variable names)*.* The number of class instances variables encapsulated by a class object is equal to the size of the complete class instance variable set of the object's class definition. There is a one-to-one correspondence between the members of the complete class instance variable set and the class instance variables of a class object. When a class object is created all of the class instance variables initially have the value of the reserved identifier "nil".

Rationale

Implementations may need to have built-in class instance variables (e.g., methodDictionary, className, etc.). Traditionally these are inherited from implementation specific classes such as Behavior or Class.

The <<class variable names>> production defines the names of discrete variables which are accessible by both class and instance methods of the class and its subclasses. The identifiers specified by <<class variable names>> are called *class variable names*. It is erroneous for the same identifier to occur more than once in the list of class variable names. The meaning of the class definition is undefined if any of the class variable names is the same identifier as an instance variable name, class instance variable name, or class variable name defined by any superclass. It is erroneous for an class variable name to also be an instance variable name or a class instance variable name.

One discrete variable, called a *class variable*, exists corresponding to each class variable name. Each class variable name is bound to the corresponding class variable. For each <<class definition>> these bindings exist in a scope called the *class variable scope*. The initial value of a class variable is the value of the reserved identifier nil. Each <<class definition>> also defines a

scope called its *inheritable class variable scope*. The inheritable class variable scope for a <<class definition>> is a composite scope whose outer scope is the inheritable class variable scope of its superclass (or the empty scope if no superclass was specified) and whose inner scope is its class variable scope.

The <<imported pool names>> production, specifies variable pools whose elements may be referenced from within <method definition> or <initializer definition>clauses that are part of the <<class definition>>. The imported pools identifiers are called *pool names*. It is erroneous if each pool name is not the <<pool name>> of a <<pool definition>> whose binding exists in the global scope of this program. It is erroneous for the same identifier to occur more than once in the list of pool names. Each class defines a scope, called its *pool variable scope*, containing the union of the names of all pool variables defined by all of the pool definitions named by the <<imported pool names>>clause. The binding of each name in the pool variable scope is the binding of the name in its corresponding <<pool definition>> unless a name is defined in more than one <<pool definition>>. In that case the binding of the name in the pool variable scope is the error binding.

3.3.2.3 Behavior Specification

A <<class definition>> defines a scope, called the *class scope*, that is used by all <method definition> and <initializer definition> clauses that are part of the <<class definition>>. The class scope for a class, X, is defined as follows:
((global scope + X's pool variable scope) + X's class variable scope.
Its *instance function scope* is defined as:
X's class scope + X's instance variable scope
and its *class function scope* is defined as:
X's class scope + X's class instance variable scope.
Where the instance variable scope is a scope that binds the elements of the complete instance variable set and the class instance variable scope is a scope that binds the elements of the complete class instance variable set.

The productions <<instance methods>> and <<class methods>> are used to specify, respectively, the instance behavior and the class behavior defined by a class definition. Each <method definition> specifies a method selector. It is erroneous if more than one <method definition> for a specific method selector appears in the <<instance methods>> of a <<class definition>>. Similarly, it erroneous if more than one <method definition> for a specific method selector appears in the <<class methods>> of a <<class definition>>.

The instance behavior defined by the class definition consists of the instance behavior (including inherited behavior) of the superclass augmented by the <<instance methods>> of the class definition. A <method definition> in the <<instance methods>> whose method selector is the same as the method selector of a <method definition> in the superclass' instance behavior replaces the inherited <method definition> in the instance behavior. The <method definition> that is replaced is called an *over-ridden method*. Similarly, the class behavior defined by the class definition consists of the class behavior of the superclass augmented in an analogous manner by the <<class methods>> of the class definition.

If the <<superclass name>> is absent then this class has no inherited instance behavior and the instance behavior consists solely of the <<instance methods>> that are part of the class definition. The class behavior of such a class is defined to inherit from the instance behavior of the <<class definition>> whose <<class name>> is the identifier "Object" bound in the global scope.

The effect of defining a method whose method selector is one of the following restricted selectors is undefined except for their use in the definition of behaviors that are required by this standard. Implementations may disallow the definition of methods with these selectors.

Restricted Selectors:	
ifTrue:	ifTrue:ifFalse:
ifFalse:	ifFalse:ifTrue:
to:do:	to:by:do:
and:	or:
==	timesRepeat:
basicAt:	basicAt:put:
basicSize	basicNew:

Rationale

Smalltalk implementations have traditionally open-coded certain messages including those with the above selectors. Open coding is typically based upon assumptions about the typical class of the receiver of these messages. If the receiver is a literal or block constructor, these assumptions can be verified at compilation time in order to make the decision as to whether open-coding is appropriate. If the receiver is a variable or expression it is difficult or impossible to verify this at compilation time. In this situation most implementations make the assumption that the receiver will be an instance of a class for which the open-coding is valid. A run-time check verifies that the receiver is an instance of the expected class and generates a run-time error if it is not. This error check precludes the polymorphic use of messages that are open coded in this manner. While this standard neither requires nor encourages this implementation technique it does allow it. This is the reason for the restrictions on the above selectors. These are the selectors that have been traditionally open-coded and whose receivers are often expressions or variables. Other messages such as #whileTrue: have also been traditionally open-coded but typically their receivers are block constructors. Thus, it is feasible to only open-code messages with compile-time verifable receivers and there is no conflict with the polymorphic use of the messages.

The <<class initializer>> is the initializer for the class. The <<class initializer>> production consists of an <initializer definition> that defines the function that is used to generate the initial values of class variables and the class object defined by the class definition. The outer scope of the class initializer is the same as the outer scope of the class methods of the same <<class definition>>. A class initializer is not inherited by subclasses. The value returned by a class initializer is discarded.

3.3.3 Global Variable Definition

A global variable definition is used to specify a discrete variable or named object that is bound to a variable name within the global definition scope. The definition may include an initializer that provides the initial value of the variable.

```
<<global definition>> ::=
                [<<constant designator>>] <<global name>> [<<variable initializer>>]
<<global name>> ::= identifier
<<variable initializer>> ::= <initializer definition>
```

If the <<*constant designator*>> is present the <<global definition>> defines a named object, otherwise it defines a global variable. The <<global name>> is the global name of the discrete variable or named object. It is erroneous if there is any other global definitions of this name within the program. It is erroneous if the <<global name>> is one of the reserved identifiers, "true", "false", "nil", "self" or "super".

The <<variable initializer>> when evaluated provides the value of the named object or the iinitial value of the global variable. If no <<variable initializer>> is present its value is nil. If the <<*constant designator*>> is present the binding of the <<global name>> to the object that is the value of the <<variable initializer>> is a constant binding and it is erroneous for an identifier that resolves to that binding to appear as the target of an assignment statement. If the <<*constant designator*>> is present the value of the named object is undefined prior to the evaluation of its <<variable initializer>>.

3.3.4 Pool Definition

A pool definition introduces a global name binding for a variable pool and defines the names of the discrete variables within the pool.

```
<<pool definition>>  ::= <<pool name>> <<pool variable definition>>*
<<pool name>> ::= identifier
<<pool variable definition>>  ::=
        [<<constant designator>> <<pool variable name>> [<<variable initializer>>]
<<pool variable name>> ::= identifier
```

The <<pool name>> is the global name of the variable pool. It is erroneous if any other global definition of this name exists within the program. It is erroneous if the <<pool name>> is one of the reserved identifiers, "true", "false", "nil", "self" or "super". An identifier that is bound to a variable pool with a <<pool definition>> is called a *pool name*. Pool names are listed in the <<imported pools>> production of a <<class definition>>. The use of a pool name in any other context is undefined.

Rationale

An implementation is permitted but not required to treat a pool name as a binding on an implementation artifact such as a pool dictionary.

A <<pool variable definition>> introduces a name binding within a variable pool for a named or object discrete variable with an optional initial value. If the <<*constant designator*>> is present the <<pool variable definition>> defines a named object, otherwise it defines a variable, called a *pool variable*. The <<pool variable name>> is the name of the pool variable or named object. It is erroneous to have more than one <<pool variable definition>> with the same <<pool variable name>> within the same variable pool. It is erroneous if the <<pool variable name>> is one of the reserved identifiers, "true", "false", "nil", "self" or "super". The <<variable initializer>>, when evaluated, provides the initial value of the pool variable or named object. If no <<variable initializer>> is present the initial value is nil. The value of a named object, before evaluation of its <<variable initializer>> is undefined. If the <<*constant designator*>> is present the binding of the <<*pool variable name*>> to the value computed by the <<*variable initializer*>> is a constant binding and it is erroneous for an identifier that resolves to that binding to appear as the target of an assignment statement.

3.3.5 Program Initializer Definition

A program initializer definition is used to specify a initializer that is executed solely for its side-effects. The value of such an initializer is not captured as the value of a variable or as a named object.

```
<<program initializer definition >> ::= <initializer definition>
```

The value of the initializer is discarded.

3.4 Method Grammar

The method grammar defines Smalltalk's language for describing units of executable code. There are three fundamental constructs that define executable code: methods, initializers, and blocks. These are generically called *functions* because they perform computations and return a result value. The statements in a function are executed when the function is activated during execution of the Smalltalk program. Each type of function is activated in a different manner. Most commonly, functions are activated as the result of some action in an already active function. In this case, the function which performs the action is known as *the calling function* and it is said to *call* the newly activated function. The definition of each type of function specifies the circumstances under which

a function of that type is activated. The next section defines the traits that are common to all types of functions.

3.4.1 Functions

The definition of any function may include a set of *temporary variable names*. In addition, the definitions of methods and blocks may also include a set of *argument names*. A function defines a scope, called its *local scope*, whose names consist of its argument names and its temporary variable names. Temporary variable names are bound to discrete variables. The bindings of argument names are constant bindings. It is erroneous for an identifier that resolves to an argument to be the target of an assignment statement.

The identifiers referenced from the statements of a function are resolved in the context of a composite scope called the *statement scope*. The inner scope of a statement scope is its function's local scope. The nature of the outer scope varies for each type of function.

When a function is activated, an individual discrete variable, called a *temporary variable*, is created corresponding to each of the function's temporary variable names. Each temporary variable is bound to the corresponding temporary variable name in the function's local scope. The value of each temporary variable is initialized to nil.

If the function requires arguments the constant binding of each argument name in the local scope is set to reference the object that is the corresponding actual argument passed by the expression that called the function.

The state of an executing function includes the active identifier bindings used by the function and the current locus of execution within the function. The noun *activation* is used to describe the state of an executing function. Each time the execution of a function is initiated a new activation is comes into being. The activation exists at least until execution of the function is irrevocably terminated. During an execution of a function, the evaluation of an expression may result in the calling of another function. When this occurs, the activation of the calling function is suspended at the current point of execution and a new activation is created for the called function. Should the called function return (complete execution), the suspended activation of the calling function is reactivated and execution resumes from the point of suspension. A called function will itself be suspended if it calls another function (or itself, recursively). Thus a logical chain of suspended activations exists that begins with some initial calling activation and proceeds through all suspended activations that lead to the activation of the currently active function. Such a chain of activations is called a *call chain*.

There are unique bindings of a function's temporary variables and arguments for each activation of the function. If a function has multiple simultaneous activations, each activation has an independent set of temporary variables and local bindings. Each temporary variable or argument binding created by a function's activation must continue to exist, retaining its value, for at least as long as it is possible for execution to reach any statement that contains an identifier whose binding resolves to a reference to the variable or argument. In most cases temporary variables can be destroyed when the activation of the function that created them is terminated. However, if a function evaluates a block constructor that results in a block that references any of the function's local variables, then those variables must continue to exist, maintaining their values, (even after the function is terminated) as long as the block or any activation of the block exists.

3.4.2 Method Definition

A method is a function that is activated as the result of sending a message to an object. A <method definition> is a component of a <<class definition>> that introduces a method along with an associated method selector into a behavior. A method consists of a sequence of statements that are evaluated when the method is activated by a message send. Evaluation of a method concludes by returning an object reference as the value of the message send that activated the method.

```
<method definition> ::=
        <message pattern>
        [<temporaries> ]
        [<statements>]
<message pattern> ::= <unary pattern> | <binary pattern> | <keyword pattern>
<unary pattern> ::= unarySelector
<binary pattern> ::= binarySelector <method argument>
<keyword pattern> ::= (keyword  <method argument>)+
<temporaries> ::= '|' <temporary variable list> '|'
<temporary variable list> ::= identifier*
```

It is erroneous if the same identifier is used for more than one <method argument> in an individual <method definition>. It is erroneous if any of the reserved identifiers ('nil', 'true', 'false', 'self', and 'super') is used as a <method argument>. It is erroneous if the same identifier is used as a <method argument> of a <method definition> and also appears in the method's <temporary variable list>. An identifier that is used as a <method argument> is called a *method argument name*.

It is erroneous if the same identifier appears more than once in a single method definition's <temporary variable list>. It is erroneous for any one of the reserved identifiers ('nil', 'true', 'false', 'self' and 'super') to appear in a <temporary variable list>. An identifier that appears in a <temporary variable list> of a method is called a *method temporary variable name*.

A <temporary variable list> list may be empty, containing no identifiers. In this case the enclosing vertical bars may be immediately adjacent with no intervening white space.

Each <method definition> has an identifying method selector. If the <method definition> has a <unary pattern> or a <binary pattern> then its method selector is the specified unarySelector or binarySelector. If the <method definition> has a <keyword pattern> then its method selector is the keywordSelector formed by concatenating, in left to right order, each keyword specified in the <keyword pattern>.

If a method is an instance method then the outer scope of its statement scope is its class definition's instance function scope and each instance variable name is bound to its corresponding instance variable of the object that is the receiver of the message that activated the method.

If a method is a class method then the outer scope of its statement scope is its class definition's class function scope and each class instance variable name is bound to its corresponding class instance variable of the class object that is the receiver of the message that activated the method. Note that because of inheritance this is not necessarily the class object defined by the <<class definition>> that defined the method.

During activation of a method the reserved identifier 'self' has a constant binding to the object that was the receiver of the message that activated the method.

The evaluation of a method is terminated either when it executes the last statement in its <statements> or by executing a return statement. If the method is terminated by a return statement then the value of the method is the value of the return statement. Otherwise the value of the method is the current binding of the reserved identifier 'self'.

3.4.3 Initializer Definition

An initializer is a function that is executed to provide an initial value for a program element. An initializer consists of a sequence of statements that are executed in sequence. The value of the initializer is used to initialize any associated variables.

```
<initializer definition> ::=
```

```
        [<temporaries>]
        [<statements>]
```

The outer scope of an initializer's statement scope varies according to the usage of the <initializer definition>. If an initializer is part of a <<global definition>> or a <<program initializer definition>> its outer scope is the global scope. If an initializer is a <<class initializer>> its outer scope is the class function scope of the <<class definition>> that includes the initializer. If an initializer is a <<variable initializer>> of a <pool variable definition> then its outer scope is a composite scope defined as:

global scope + pool scope

where pool scope is a scope that binds each <pool variable name> of the pool to its associated discrete variable or named object.

During activation of an initializer the reserved identifier 'self' has the error binding unless the initializer is a <<class initializer>>. During activation of a <<class initializer>> the binding of 'self' is the class object of the <<class definition>> that defines the <<class initializer>>.

The evaluation of an initializer is terminated either when it executes the last statement in its <statements> or by the execution a <return statement> during the evaluation of any block that is created from a <block constructor> contained within the initializer's <statements> during the activation of the initializer. If the initializer is terminated by a <return statement> then the value of the initializer is the value of the <return statement>. Otherwise the value of the initializer is the value of its last statement. The value of an initializer with no <statements> is the binding of the reserved identifier 'nil'.

3.4.4 Blocks

A block is a function that can be manipulated as an object that implements the <valuable> protocol. A block is always defined, using a <block constructor>, as a <primary> expression element within another function, called its *enclosing function*. Thus a block is always nested within a method, an initializer, or another block. Blocks may be nested to an arbitrary level. The outermost function enclosing a block is called the *home function* of the block. A block object is created when the <block constructor> for the block is evaluated in the course of executing the block's enclosing function. If the enclosing function of a block is a method or an initializer the *home activation* of the block is the activation of its enclosing function that created the block. If the enclosing function of a block is a block then the home activation of the block is the same as the home activation of the block that created the block.

The <statements> of a block are normally evaluated when the block is activated by sending a variant of the #value message to the block object. However, other methods defined in this standard also specify that they cause the evaluation of blocks. In these cases, evaluation of the block proceeds as if a #value message variant had been sent to the block. Evaluation of a block normally concludes by executing the last statement of the block. In this case the object reference that is the value of the last statement is returned as the value of the message send that activated the block. If the last statement of a block is a <return statement> evaluation of the block's home activation is also terminated and the value of the <return statement> is used as the return value of the home activation.

Expressions within a block may reference temporary variables and arguments of the functions that enclose the block. Each block object is an independent *closure* that captures the current bindings for any enclosing functions' arguments or temporaries that are referenced from within the block's <block constructor>. Any such captured bindings and their associated discrete variables or objects must be preserved as long as the block object continues to exist and is available for evaluation. Note that the values of any such captured discrete variables and the state of any object captured by an argument binding remain subject to possible modification.

```
<block constructor> ::= '[' <block body> ']'
<block body> ::= [<block argument>* '|'] [<temporaries>] [<statements>]
```

```
<block argument> ::= ':'  identifier
```

If the <block body> does not have any <block argument> clauses then the objects that are the value of the <block constructor> conform to the protocol <niladic-block>. If the <block body> has exactly one <block argument> then objects that are the value of the <block constructor> conform to the protocol <monadic-block>. If the <block body> has exactly two <block argument> clauses then objects that are the value of the <block constructor> conform to the protocol <dyadic-valuable>. If the <block body> has more than two <block argument> clauses then objects that are the value of the <block constructor> conform to the protocol <valuable>.

If any block arguments are present, the final block argument is followed by a vertical bar ("|"). If a <temporaries> clause is present then the first temporary variable is preceded by a vertical bar. A vertical bar that terminates a sequence of block arguments may be immediately adjacent (with no intervening white space) to the vertical bar that initiates a <temporaries> clause.

It is erroneous if the same identifier is used for more than one <block argument> of a individual <block constructor>. It is erroneous for any one of the reserved identifiers ('nil', 'true', 'false', 'self' and 'super') to be used as a <block argument>. It is erroneous if the same identifier is used both as a <block argument> and also appears in the <temporaries> of a single <block constructor>. An identifier that is used as a <block argument> is called a *block argument name*. An identifier that appears in the <temporaries> of a <block constructor> is called a *block temporary variable name*.

The outer scope of a block's statement scope is the statement scope of the block's enclosing function. Within a <block constructor> the binding of the reserved identifier 'self' is the same binding as the binding of 'self' for the block's home activation.

If a block has no <block body> or no <statements> in its <block body> then the value of the block is undefined.

3.4.5 Statements

When a function is activated the expressions defined in the <statements> portion of the function's definition are evaluated. Each such expression is called a *statement*.

```
<statements> ::=
        (<return statement> ['.'] ) |
(<expression> ['.' [<statements>]])
```

<statements> consists of a sequence of statements. Each statement except the final statement of the sequence is an <expression>. The last statement in a <statements> sequence may be either an <expression> or a <return statement>. Each <expression> within a <statements> is separated from its following statement by a period ('.'). A period is optional following the last statement.

The individual statements are evaluated in left to right sequence. All identifiers within the statements are resolved using the *statement scope* of the immediately enclosing function. Identifiers within block constructors are resolved using the block constructor's *statement scope*. The value returned by each statement except, in some circumstances, the last statement is discarded.

3.4.5.1 Return statement

If the last <expression> in a <statements> clause is proceeded by a circumflex ('^') the <expression> forms a return statement and the value computed by the expression is the value of the return statement.

```
<return statement> ::= returnOperator  <expression>
```

A return statement returns the value of it's <expression> as the value of the method or initializer in which it appears.

If a return statement is the last statement of a block, execution returns from the home activation of that block and the value of the return statement becomes the value returned from its home activation. It is undefined to execute a return statement from a block activation if the home activation of that block has already returned a value or has otherwise terminated. It is undefined to execute a return statement from a block activation if the block's home activation does not exist on the call chain that leads to the block activation.

If the home activation is an initializer activation the value of the return statement becomes the value of the initializer and execution proceeds with the evaluation of the next initializer in the global initialization sequence.

If the home activation is not an initializer execution proceeds by resuming execution of the function activation that was suspended when the home activation was created. The value of the return statement becomes the value of the message that resulted in the creation of the home activation.

Execution of a return statement within a block results in the abnormal-termination of any suspended function activations that exist on the call chain leading from the block's home activation to the block action executing the return statement. If a function activation that is abnormally terminated by a return statement is a block activation that was created in the course of evaluating the receiver block of an #ensure: or #ifCurtailed: message then the termination block argument of the #ensure: or #ifCurtailed: message is evaluated prior to completion of the return statement.

The evaluation of any such termination blocks occurs as if the message #value had been sent to the termination block. The evaluation of termination blocks occurs subsequent to the evaluation of the return statement's expression but prior to the return of any value from the home activation. If there are multiple termination blocks on the call chain, they are evaluated starting with the termination blocks that most closely precedes, on the call chain, the activation executing the return statement and continuing in reverse order of their occurrence on the call chain. If the evaluation of a termination block concludes with the execution of a return statement the result is undefined. The result is also undefined if evaluation of the termination block results in evaluation of any block that concludes with a return statement and whose home activation is not on the call chain that starts with the activation of the termination block.

3.4.5.2 Expressions

Statements are composed of expressions. An expression is a sequence of tokens that describes a reference to an object or a computation that produces a reference to an object. The resultant object is called the *value* of the expression. An expression may optionally specify that its value is to be assigned to one or more variables. The primary constituent of an expression is a variable, named object, literal, block constructor, or a parenthesized subexpression. The primary either directly provides the value of the expression or serves as the receiver of a set of messages that compute the value of the expression.

```
<expression> ::=
<assignment> |
<basic expression>
<assignment> ::= <assignment target> assignmentOperator  <expression>
<basic expression> ::=
        <primary> [<messages> <cascaded messages>]
<assignment target> := identifier
<primary> ::=
        identifier |
```

```
        <literal> |
        <block constructor> |
        ( '(' <expression> ')' )
```

An <assignment target> is a variable name that is called the *target* of the assignment. The value of the <expression> to the right of the assignmentOperator replaces the current value of the <assignment target> variable. The target must have a binding to a variable in the statement scope that contains the <expression>. It is erroneous if a binding for the <assignment target> identifier does not exist in the statement scope. It is erroneous if the binding of the target is a constant binding. It is erroneous if target is one of the reserved identifiers: 'true', 'false', 'nil', 'self', 'super'.

An <assignment> may assign its value to multiple target variables by including multiple assignmentOperator clauses. The value of an <assignment> expression is the value that is assigned to its target variable. All target variables in an assignment with multiple targets are assigned the same value.

A <primary> is the basic unit from which expressions are constructed. A <primary> that consists of an identifier is a reference to the value of a variable, named object, or reserved identifier. The identifier must be a name that is bound in the statement scope that contains the expression. The value of such a <primary> is the value of the entity that is bound to the identifier. It is erroneous if the identifier does not have a binding in the statement scope. If the binding of the identifier is to a <<pool name>> its value is undefined. It is erroneous if a <basic expression> consists solely of the reserved identifier 'super'. 'super' may only appear if it is followed by a <messages> clause.

A <primary> that is a <literal> is a reference to a statically created object. The value of the primary is the object. The type of object is determined by the syntactic form of the literal.

The value of a <primary> that is a <block constructor> is a reference to a block object whose outer scope is the statement scope of the function that contains the <block constructor> and whose home activation is the home activation of the enclosing function. It is unspecified whether separate evaluations of a <block constructor> produce distinct objects.

Rationale

Traditionally, a block constructor always creates a new object. This is necessary if the block captures any bindings from its enclosing routines or contains a return statement. By leaving the identify of successive block constructor values undefined, we permit implementation to optimize other cases by statically creating block objects.

The value of a <primary> that is a parenthesized <expression> is the value of the <expression>.

3.4.5.3 Messages

Messages cause the activation of a method. There are three syntactic forms of message sends. They correspond to the three types of message selectors: unary, binary and keyword. Every message send has a value that is the result returned from the evaluation of its method.

```
<messages> ::=
        (<unary message>+ <binary message>* [<keyword message>] ) |
        (<binary message>+ [<keyword message>] ) |
        <keyword message>
<unary message> ::= unarySelector
<binary message> ::= binarySelector <binary argument>
<binary argument> ::= <primary> <unary message>*
<keyword message> ::= (keyword <keyword argument> )+
<keyword argument> ::= <primary> <unary message>* <binary message>*
<cascaded messages> ::= (';' <messages>)*
```

Syntactically, the three forms of <messages> are similar in that the receiver is always written first,

followed by the selector and arguments. The *receiver* of a message is the value of the <primary> or the message send to the immediate left of a message's selector. The receiver is a reference to an object. It can be represented either as a literal, an identifier, a block constructor, or another expression . All message arguments are also references to objects, represented in the same way as the receiver. The receiver and the arguments are evaluated before the message is sent. They are evaluated in a left-to-right order.

Unary messages have no arguments.

Binary messages require one argument.

A keyword message takes one or more arguments and is composed of a sequence of keywords followed by expressions. The number of keywords is equal to the number of arguments. Each keyword is used to associate the argument immediately following it with a corresponding argument of the method that is activated.

A <messages> clause can be made up of multiple message sends. The order of evaluation of the message sends are defined by the following precedence rules:. Sequences of <unary message> clauses are evaluated left to right. The result of each message becomes the receiver to the <unary message> to its immediate right. Sequences of <binary message> clauses are also evaluated strictly left to right. The <binary argument> of a <binary message> is evaluated before performing the binary message send. The result of each binary message becomes the receiver for the <binary message> to its immediate right. The <keyword argument> clauses of a <keyword message> are evaluated left to right. The final message send of a <messages> clause is its <keyword message>.

The method selector of a <unary message> is its unarySelector. The method selector of a <binary message> is its binarySelector. The method selector of a <keyword message> is formed by concatenating, in left to right order each keyword. The ordering of keywords is an essential property of keyword messages. Different orderings of a common set of keywords produce different selectors.

A message send is evaluated by locating and activating a method. The method is located by matching the message's method selector with the selectors of the methods that compose the behavior of the receiver. The method to be activated is the method whose selector is identical to the message's selector.

If a method is located, the current function activation is suspended and the selected method is activated. The bindings of 'self' and 'super' in the local scope of the newly activated method are constant bindings to the object that was the receiver of the message. Within the local scope, the identifier associated with each <method argument> in the <message pattern> of the method's definition is bound with a constant binding to the corresponding argument object. When the method completes execution, execution of the suspended activation is resumed with the value of the method serving as the value of the message that activated it.

3.4.5.3.1 Sends To 'super'

If the <primary> that defines the receiver of a message is the reserved identifier 'super' the method is located using the behaviors of the class definition that is the superclass of the class definition that includes the definition of the method containing the <primary>. If the method is an instance method, the superclass' instance behavior is used. If the current method is a class method the superclass' class behavior is used. If the method is a class method, and the class definition that defines the method does not have a superclass then the behavior to use is unspecified. A method is selected by matching the message's method selector with the methods that compose the specified behavior. The method to be activated is the method whose selector is identical to the message's selector. The meaning is undefined if the receiver is 'super' and the specified behavior does not include the definition of a method with a matching selector. It is erroneous if the receiver is 'super', the current method is an instance method, and the class that defines the method does not have a superclass.

3.4.5.3.2 Message Not Understood

If a method matching the message's selector does not exist in the behavior of the receiver the message send is a *failed send* and the following actions occur. A new object that conforms to the protocol <failedMessage> is created. It is initialized such that if sent the message #selector it returns an object that is equal to message's selector of the failed send and if sent the message #arguments it returns a sequence of objects whose elements are the arguments of the failed send. Execution then proceeds to locate a method whose selector matches the literal selector #doesNotUnderstand: in the behavior of the receiver of the failed send. It is erroneous if the receiver's behavior does not include a method with a matching selector. If a method with a matching selector is located, the method is activated with the receiver of the failed send bound to 'self' and the <failedMessage> object bound to the argument of the method.

3.4.5.3.3 Cascades

A *cascade* is a sequence of message sends that are all directed to the same object. Only the first in such a sequence has an explicit receiver specified via a <primary>. The receiver of the subsequent messages is the same object as the receiver of the initial message in the sequence. Otherwise, each message send occurs as if it was a normal message send that was not part of a cascade. The result object of each message in the cascade except the right most message is discarded. The value of a <messages> clause that includes a <cascaded messages> clause is the value of its right most message. If the <primary> that provides the receiver of the first message in a cascade consists solely of the reserved identifier 'super' then each message in the cascade performs is if it was a message with 'super' specified as its receiver.

3.4.5.3.4 Reserved Messages for Indexable Objects

If the method selector of a message is equal to the literal selector #basicAt:, the <primary> that provides the receiver consists solely of the reserved identifier 'self', and the receiver is an indexable object or a byte indexable object the following actions are performed. The argument of the message is used as a numeric index that identifies one of the receiver's unnamed instance variables. The value of the identified instance variable is returned as the value of the message send. If the receiver is a byte indexable object the returned value is an object that conforms to the protocol <integer>. It is erroneous if the value of the argument does not conform to the protocol <integer>. It is erroneous if the integer value of the argument is less than or equal to zero or if it is greater than the number of unnamed instance variables of the receiver.

If the method selector of a message is equal to the literal selector #basicAt:put, the <primary> that provides the receiver consists solely of the reserved identifier 'self', and the receiver is an indexable object or a byte indexable object the following actions are performed. The first argument of the message is used as a numeric index that identifies one of the receiver's unnamed instance variables. The value of the message's second argument is assigned to the identified instance variable and is also returned as the value of the message send. It is erroneous if the value of the first argument does not conform to the protocol <integer>. It is erroneous if the numeric value of the first argument is less than or equal to zero or if it is greater than the number of unnamed instance variables of the receiver. If the receiver is a byte indexable object it is erroneous if the value of the second argument is not an object that conforms to the protocol <integer> and whose value is in the range 0 to 255.

If the method selector of a message is equal to the literal selector #basicSize, the <primary> that provides the receiver consists solely of the reserved identifier 'self', and the receiver is an indexable object or a byte indexable object the following actions are performed. An object that conforms to the protocol <integer> is returned as the value of the message send. The numeric value of the object is equal to the number of unnamed instance variables of the receiver. If the receiver has no unnamed instance variables the numeric value of the returned object is zero.

If the method selector of a message is equal to the literal selector #basicNew:, the <primary> that

provides the receiver consists solely of the reserved identifier 'self', and the receiver is the class object of a class whose instance objects are indexable or byte indexable the following actions are performed. A new instance of the receiver is created that has the number of unnamed instance variables that is specified by the value of the argument to the message. The new object is returned as the value of the message send. It is erroneous if the value of the argument does not conform to the protocol <integer>. It is erroneous if the integer value of the argument is less than zero. The result is undefined if it is impossible to create an object of the size specified by the argument.

3.4.6 Literals

A literal is a syntactic construct that that directly describes a statically created object. Instances of several classes of objects can be represented literally. These include numbers, characters, strings, symbols, message selectors, and arrays. Each type of literal is discussed in individual sections below. For each type of literal, a protocol is specified to which objects of that literal form must conform.

```
<literal> ::=
        <number literal> |
        <string literal> |
        <character literal> |
        <symbol literal> |
        <selector literal> |
        <array literal>
```

The protocols specified for literals do not include any messages that modify the state of the literal objects. The effect of sending a message to an object that is the value of a literal that modifies the state of the literal is undefined.

Multiple identical literals may occurs within a Smalltalk program. It is unspecified whether the values of identical literals are the same or distinct objects. It is also unspecified whether the values of separate evaluations of a particular literal are the same or distinct objects.

3.4.6.1 Numeric Literals

Numbers are objects that represent numerical values. Numeric literals are used to create numeric objects which have specific values and numeric representations.

```
<number literal> ::= ['-'] <number>
<number> ::= integer | float | scaledDecimal
```

If the preceding '-' is not present the value of the numeric object is a positive number. If the '-' is present the value of the numeric object is the negative number that is the negation of the positive number defined by the <number> clause. White space is allowed between the '-' and the <number>.

If the <number> clause is an integer the value of the literal is an object that responds to the <integer> protocol and whose value represents the numeric value of the integer. Integer objects correspond to ISO/IEC 10967 integers with unbounded range. There is no maximum magnitude for an integer.

If the <number> is a float the value of the literal is an object that responds to the <Float> protocol. The maximum precision of a float is implementation-defined. If the number of digits in the mantissa of the float exceeds the maximum precision then the mantissa will be rounded to the maximum.

An implementation may support up to three different floating point numeric representations with varying precision and ranges. The floating point numeric representations are characterized by the objects that are the values of the standard globals named FloatE, FloatD, and FloatQ. These objects all conform to the protocol <floatCharacterization>and can report the values of parameters that describe the characteristics of a floating point numeric representation. If an implementation supports three floating point representations then the characterization parameters of FloatE, FloatD, and FloatQ will each be different. If an implementation supports two floating point representations then either the characterization parameter of FloatE and FloatD are equal, or the characterization parameter of FloatD and FloatQ are equal. If an implementation supports only one floating point representations then the characterization values of FloatE, FloatD, and FloatQ are all equal. One of the characteristic parameters of a floating point numeric representation is its *precision*. It is required that:

(FloatE precision) ≤ (FloatD precision) ≤ (FloatQ precision).

The numeric representation used for a floating point literal is determined by the exponentLetter if it is present in the float. If the exponentLetter is 'e' the floating point representation characterized by FloatE is its selected representation. If the exponentLetter is 'd' the floating point representation characterized by FloatD is its selected representation. If the exponentLetter is 'q' the floating point representation characterized by FloatQ is its selected representation.

If a floating point literal does not include an explicit exponentLetter its selected representation is the floating point representation with the smallest precision that can represent the numeric value of the float with no loss of precision or, if no such representation exists, the representation with the greatest precision.

The value of the floating point object is the value using the selected floating point representation that most closely approximates the numeric value of the float. If the number of digits in the mantissa of the float exceeds the maximum precision of the selected representation the mantissa will be rounded to the representation's maximum precision. It is erroneous if the numeric value defined by float is outside the range of values expressible using the selected representation.

If the <number> is a scaledDecimal the value of the literal is a numeric object that responds to the <scaledDecimal> protocol. Scaled decimal objects provide a precise representation of decimal fractions with an explicitly specified number of fractional digits. The specified number of fractional digits in the scaled decimal object is the greater of the numeric value of fractionalDigits and the actual number of digits to the right of the decimal point in the scaledMantissa. It is erroneous if the numeric value of fractionalDigits is smaller than the actual number of digits, if any, to the right of the decimal point in the scaledMantissa.

The maximum allowed precision for a scaled decimal numeric object is implementation defined and may be unbounded. It is erroneous if the total number of digits including the specified number of fractional digits exceeds the implementation defined maximum precision.

3.4.6.2 Character Literals

Character literals define objects that represent individual symbols of an alphabet. Characters are most commonly used as the elements of strings.

<character literal> ::= quotedCharacter

The value of a character literal is an object that conforms to the <character> protocol. It is erroneous if the character part of the quotedCharacter does not exist in the implementation defined execution character set used in the representation of character objects.

3.4.6.3 String Literals

String literals define objects that represent sequences of characters.

```
<string literal> ::= quotedString
```

The value of a string literal is an object that conforms to the <readableString> protocol. The elements of the object consist of objects representing the individual characters that make up the stringBody. For the purpose of defining the string each individual character is treated as if it was the character of a <character literal>. Any paired stringDelimiter characters within the stringBody are treated as one character object that encodes the string delimiter character.

It is erroneous if stringBody contains any characters that does not exist in the implementation defined execution character set used in the representation of character objects.

If the stringBody is not present the value of the string literal is a <readableString> object containing no characters. Its size is zero.

3.4.6.4 Symbol Literals

Symbols are strings that are identity objects.

```
<symbol literal> ::= hashedString
```

The value of a symbol literal is an object that implements the <symbol> protocol. The elements of the object consist of objects representing the individual characters that make of the stringBodyof the hashedString. For the purpose of defining the symbol each individual character is treated as if it was the character of a <character literal>. Any paired stringDelimiter characters within the stringBody are treated as one character object that encodes the string delimiter.

It is erroneous if stringBody contains any characters that do not exist in the implementation defined execution character set used in the representation of character objects.

If the stringBody is not present the value of the symbol literal is the unique <symbol> object containing no characters. Its size is zero.

Symbol objects are identity objects. If two symbols are equal they are the same object. Two symbol literals with identical stringBody parts evaluate to the same symbol object. Every evaluation of a particular <symbol literal> always returns the same object.

3.4.6.5 Selector Literals

Selectors are objects which may be used as method selectors in perform messages.

```
<selector literal> ::=  quotedSelector
```

The value of a selector literal is an object that implements the <selector> protocol. Selector objects represent method selectors and can be used in conjunction with perform messages to dynamically send messages.

Selector objects are identity objects. If two selectors are equal they are the same object. Two selector literals with identical quotedSelectors will evaluate to the same symbol object. Every evaluation of a particular <selector literal> always returns the same object.

Some implementations may wish to implement selector objects such that they conform to both

<selector> protocol and <symbol> protocol. It is implementation defined whether a symbol literal whose stringBody is identical to the selectorBody of a selector literal evaluates to the same object as the selector literal.

Rationale

Because symbols and selectors are defined as supporting different protocols and because the standard does not define any messages that generate selectors from text strings it is possible to build an implementation that can analyze the program text to determine which methods have selectors that are not used by the program. Alternatively, a traditional implementation where selectors and symbols are equivalent constructs is also permitted.

3.4.6.6 Array Literals

An array literal is a sequenced collection with numeric keys which may contain any number of other literals.

```
<array literal> ::= '#(' <array element>* ')'
<array element> ::= <literal> | identifier
```

The value of an array literal is an object that implements the <sequencedReadableCollection> protocol. The elements of an array literal can consist of any combination of literal forms. If an identifier appears as an <array element> and it is one of the reserved identifiers nil, true or false the value of the corresponding element of the collection is the value of that reserved identifier. The meaning is undefined if any other identifier is used as an <array element>. If an <array literal> has no <array element>clauses the collection has no elements.

3.4.7 Reserved Identifiers

The following identifiers are reserved words in Smalltalk. They may only be used as a <primary> and are defined as follows:

nil	A constant binding to a unique object that supports the <nil> protocol. The scope of the binding is the entire program. Variables that have not been explicitly initialized initially have this value.
true	A constant binding to a unique object that supports the <boolean> protocol. The scope of the binding is the entire program.
false	A constant binding to a unique object that supports the <boolean> protocol. The scope of the binding is the entire program.
self	Within a method, a constant binding to the receiver of the message that activated the method. The scope of the binding is a single method activation. Within a <<class initializer>> it is a constant binding to the associated class object. Within any other type of initializer self has the error binding.
super	Within a method, a constant binding to the receiver of the message that activated the method. The binding of 'super' is to the same object as the binding of 'self', but causes message lookup to start in the superclass of the class containing the method in which super appears, rather than starting in the class of the receiver. The major purpose of a message to super is to invoke a method in a superclass which is over-ridden in a subclass. Super must be followed by a message send. It cannot be used in place of self as a value. Within any type of initializer super has the error binding.

The objects that are the values of "nil", "true", and "false" must be distinct from one another.

The use of these reserved identifiers in any other context is erroneous .

Implementations may define other identifiers with bindings that have implementation specified semantics. Any such identifier must be bound in the extension scope of the program.. An explicit definition of such an identifier in any scope supersedes the implementation provided binding.

3.5 Lexical Grammar

The lexical grammar defines the syntax of the atomic symbols, called tokens, used in the method grammar and program grammar. Tokens are ordered sequences of characters. A character is the smallest possible syntactic unit of the token grammar. Each token is to be recognized as the longest string of characters that is syntactically valid, except where otherwise specified. Unless otherwise specified, white space or another separator must appear between any two tokens if the initial characters of the second token would be a valid extension of the first token. White space is not allowed within a token unless explicitly specified as being allowed.

3.5.1 Character Categories

The tokens of the concrete syntax are composed from an alphabet of characters. This standard does not specify the use of a particular character set or encoding. An implementation must specify its specific character set and its encoding. All implementations must support the following categories of characters:

The lowercase letters of the English alphabet.
The uppercase letters of the English alphabet.
The Arabic numerals.
A specific set of binary operators and other special characters.
A set of characters that represent "white space".

An implementation may define characters in addition to those listed below in each character category. While the meaning of a program that uses any such characters is well defined it may not be portable between conforming implementations.

```
character ::=
        "Any character in the implementation-defined character set"

whitespace ::= "Any non-printing character interpreted as white space including spaces, tabs, and
line breaks"

digit ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

uppercaseAlphabetic ::=
'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' |
'N' | 'O' | 'P' | 'Q' | 'R' | 'S'| 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z'
lowercaseAlphabetic ::=
'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'I' | 'j' | 'k' | 'l' | 'm' |
'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z'

nonCaseLetter ::= '_'
 letter ::=
        uppercaseAlphabetic |
        lowercaseAlphabetic |
        nonCaseLetter |
        "implementation defined letters"
```

3.5.2 Comments

Comments exist to allow the programmer to add documentation to a program

```
commentDelimiter ::= ""
nonCommentDelimiter::=
        "any character that is not a commentDelimiter "
 comment :=
        commentDelimiter  nonCommentDelimiter * commentDelimiter
```

The quote character " begins and ends a comment. Comments do not nest. A comment is considered white space and acts as a separator. There is no need to allow embedded quote characters in comments as this double-quote construct can simply be parsed as two sequential comments. White space characters are allowed within comments

3.5.3 Identifiers

Identifiers are used to name entities defined by the program or implementation such as variables, classes, message selectors.

```
identifier ::= letter (letter | digit)*
```

An identifier is an sequence of letters and digits. The sequence may be of any length. The first character must be a letter. Upper-case and lower-case letters are logically different. Identifiers starting with an underscore ("_") are reserved for use by the implementation. Implementations may define additional nonCaseLetter and letter characters but their usage is non-portable.

3.5.4 Keywords

Keywords are identifiers used to create message selectors.

```
keyword ::= identifier ':'
```

Keywords are identifiers followed immediately by the colon character. An unadorned identifier is an identifier which is not immediately preceded by a '#'. If a ':' followed by an '=' immediately follows an unadorned identifier, with no intervening white space, then the token is to be parsed as an identifier followed by an assignmentOperator not as an keyword followed by an '='.

3.5.5 Operators

Three types of operator tokens are defined in Smalltalk: binary selectors, the return operator, and the assignment operator.

.

```
binaryCharacter ::=
        '!' | '%' | '&" | '*' | '+' | ',' | '/' | '<' | '=' | '>' | '?' | '@' | '\' | '~' | '|' | '-'
binarySelector ::= binaryCharacter+

returnOperator ::= '^'

assignmentOperator ::=  ':='
```

Binary selectors are method selectors that appear similar to mathematical operators. A binary selector may be any length greater than or equal to one. If a negative <number literal> follows a binary selector there must intervening white space.

An implementation may define additional binaryCharacters but their use may result in a non-portable program.

3.5.6 Numbers

Numbers are tokens that represent numeric quantities. There are three forms of numbers: integer, float, and scaledDecimal. No white space is allowed within a numeric token.

```
integer ::= decimalInteger  | radixInteger
decimalInteger ::= digits
digits ::= digit+
radixInteger ::= radixSpecifier  'r' radixDigits
radixSpecifier := digits
radixDigits ::= (digit | uppercaseAlphabetic)+
```

Integer tokens describe whole numbers using any radix between 2 and 36. If no radix is specified then the radix is 10. The radixSpecifier is interpreted as a decimal integer whose numeric value must be in the range $2 \leq radixSpecifier \leq 36$. The digits used to form a radix number are the numerical digit characters and the upper case alphabetic characters. The uppercase alphabetic characters represent the digits with values 10_{10} through 35_{10} where 'A' represents the digit value 10_{10}, 'B' represents 11_{10}, and so on up to 'Z' representing the digit value 35_{10}. It is erroneous if a character representing a digit value greater than or equal to or the numeric value of the radixSpecifier is used to form the radixDigits. It is erroneous if the numeric value of the radixSpecifier is less than 2 or greater than 36.

```
float ::= mantissa [exponentLetter exponent]
mantissa ::= digits '.' digits
exponent ::= ['-']decimalInteger
exponentLetter ::= 'e' | 'd' | 'q'
```

Floating-point tokens represent numbers in scientific notation. The mantissa contains the significant digits of the number and the exponent defines a power of ten that the mantissa is to be multiplied by to obtain the numerical value of the float. Both the mantissa and exponent are written in decimal notation. The mantissa must contain a decimal point ('.'). The decimal point must be preceded and followed by at least one digit. If the exponent contains a '-' the numeric value of the exponent is a negative number. The numeric value of a float is the numeric value of the mantissa multiplied by ten raised to the power specified by the numeric value of the exponent. If the optional exponent is not present the value of the float is simply the numerical value of the mantissa. An exponentLetter must be followed by an explicit exponent

Rationale

Constants such as 1.0q are not valid. This is to avoid ambiguity relating to a minus ('-') operator immediately following such a token.

The exponent Letter is only used in floating-point numbers. Historically some implementations have allowed its use in the specification of integer constants. For example 10e10 is not a valid number token. It has not been universally implemented and its utility was deemed insufficient to justify the effort to define and implement itl.

```
scaledDecimal ::= scaledMantissa 's' [fractionalDigits]
scaledMantissa ::= decimalInteger | mantissa
fractionalDigits ::= decimalInteger
```

ScaledDecimal tokens describe decimal fractions that are to be represented using a specified number of fractional decimal digits of precision. The scaledMantissa specifies the decimal numeric value of the number. If fractionalDigits is present it specifies that the representation used for the number must allow for a number of digits to the right of the decimal point that is equal to the numeric value of fractionalDigits.

Rationale

123s = 123s0
123.0s = 123s1=123.0s1
123.000s=123s3 = 123.0s3=123.00s3=123.000s3

Allowing the digits following the 's' to be missing seems inconsistent with the rule that the exponentLetter of a float must be followed by digit

3.5.7 Quoted Character

A quoted character is a distinct token consisting of a dollar sign followed by any single character in the implementation defined character set, including a white space character

```
quotedCharacter ::= '$' character
```

3.5.8 Quoted Strings

A quoted string token is a delimited sequence of any characters in the implementation defined character set.

```
quotedString ::= stringDelimiter stringBody stringDelimiter
stringBody ::= (nonStringDelimiter | (stringDelimiter stringDelimiter)*)
stringDelimiter ::= '"     "a single quote"
nonStringDelimiter ::= "any character except stringDelimiter"
```

A nonStringDelimiter is any character in the implementation defined character set except a stringDelimiter. A single stringDelimiter may be represented in the string by two successive stringDelimiter. There is no limit on the length of a quoted string token. White space characters may be included in the stringBody.

3.5.9 Hashed String

A hashed string is a quoted string that is immediately preceded by a pound sign.

```
hashedString ::= '#' quotedString
```

The stringBody of a hashedString may include white space characters.

3.5.10 Quoted Selector

A quoted selector is an identifier, binary selector or sequence of keywords that is immediately preceded by a pound sign.

```
quotedSelector ::= '#' (unarySelector  |  binarySelector  | keywordSelector)
keywordSelector ::= keyword+
```

3.5.11 Separators

```
separator ::= (whitespace | comment)*
```

Smalltalk programs are free-format. Unless otherwise specified, a separator must appear between two tokens if all or any initial part of the second may appear as a valid extension of the first. A separator may appear between any two tokens. They may not appear within a token except where

explicitly allowed to so appear. Anywhere one separator may appear an arbitrary number may appear.

3.6 Implementation Limits

The portability of a conforming program may be dependent upon the limits of numerous parameters of a conforming Smalltalk language implementation. For certain parameters, the standard defines a lower limit for all conforming implementations. Other lower limits are left unspecified by the standard. All such specified limits are minimally acceptable lower bounds. Implementation are discouraged from imposing any unnecessary restrictions on any implementation parameters.

The values of the following implementation parameters are implementation defined and must be documented by conforming implementations:

Parameter	**Minimum upper bound**
Length of identifiers	200
Length of binary selectors	2
Total length of keyword selectors (including colons)	500
Number of named instance variables per object (including inherited)	127
Number of class variables per class	127
Number of variables per pool	1000
Number of methods per behavior	1000
Number of arguments per method or block	15
Number of temporary variables per method or block	15
Float precision	unspecified
ScaledDecimal precision	30
Total number instance variables	65535

4. Smalltalk Interchange Format

This section gives a concrete syntax for the interchange of Smalltalk programs. This defines one of many possible concrete implementations of the program definition syntax given in section 3.3. This

interchange format is not a user specification language for the concrete syntax of these elements. It is only intended for program interchange.

Each interchange file consists of a set of definitions which form part or all of a Smalltalk program. Each definition represents all or part of a <<program element>> in the program definition syntax.

Interchange files are composed of units, called "chunks" , which are delimited by exclamation points. Each program element, as defined in section 3.3 is a represented by one or more chunks. Exclamation points which appear within a chunk are doubled but represent a single exclamation point. When processing an interchange file, each chunk should be sequentially read and preprocessed converting doubled exclamation points into single exclamation points. The body of the chunk can then be processed as specified by the Interchange BNF Syntax.

Rationale

A interchange format is a derivative of the code file format defined by Krasner in "The Smalltalk-80 Code File Format" in "Smalltalk-80, Bits of History, Words of Advice". The syntax was designed so that it could be implemented by reading chunks and evaluating them (although this may restrict those implementations to treat certain globals, such as Class, Global ,Pool, and Annotation and certain messages, such as #methods, #initializer, and #initializerFor:, in some restricted way which may cause a small set of Standard-compliant programs to be non-portable to those implementations).

4.1 Interchange Format BNF Syntax

The interchange BNF references the following symbols from the method grammar and lexical grammar as defined in sections 3.5, 3.4.2 and 3.3.5: comment, identifier, string, stringDelimiter, whitespace, <initializer definition>, and <method definition>. References to these symbols from the interchange BNF are to be interpreted using their section 3 definitions. The BNF is written with the construct <elementSeparator> as the break character for the chunks. Any processing to convert doubled exclamation points is assumed to be done as part of the initial scanning, and is not reflected in the BNF.

```
<interchangeFile> ::=
        <interchangeVersionIdentifier>
        ( <interchangeUnit>  ) +

<elementSeparator> ::= '!'

<interchangeUnit> ::= <interchangeElement> <annotation>*

<interchangeElement> ::=
        <classDefinition> |
        <classInitialization> |
        <globalDefinition> |
        <globalValueInitialization> |
        <poolDefinition> |
        <poolVariableDefinition> |
        <poolValueInitialization> |
        <methodDefinition> |
        <classMethodDefinition> |
        <programInitialization> |
        comment <elementSeparator>
```

An interchange file consists of a version specification followed by an ordered list of interchange elements. Each of the elements is terminated by an exclamation point. The interchange file corresponds to all or part of a <<Smalltalk program>> from the section Smalltalk Abstract Program Grammar . A complete program is treated as a concatenation of the interchange files from which it

is composed. Any names or objects that are predefined by an implementation are treated as if their definitions preceded the first file in this concatenation.

Each <interchangeUnit> is composed of lexical tokens as defined by the Smalltalk lexical grammar in section 3.5. Generally each token may be separated from the next by any amount of whitespace.

Each <interchangeElement> corresponds to all or part of a <<program element>> as defined by the Smalltalk Abstract Program Syntax. The annotations allow extra-lingual information to be associated with individual program elements. Collectively, the interchange elements correspond to the <<program element>>+ list of a <<Smalltalk program>>. The <<*initialization ordering*>> of the <<*Smalltalk program*>> is defined to be the ordering of the initializations elements in the interchange files. The elements and annotations may be separated by any amount of whitespace.

```
<interchangeVersionIdentifier> ::=
        'Smalltalk' 'interchangeVersion:' <versionId> <elementSeparator>
<versionId> ::= quotedString
```

In order to accommodate future changes in the format, each interchange file starts with a string that identifies the version of the interchange format used for that file. The version identifier is a constant string. If the <versionId> is the quotedString '1.0' the interchange file must strictly conform to the format specified in this standard. Any future revisions of this standard that extends or modifies the interchange file format will specify a different value for the quotedString. Any non-standard extensions to the interchange file format should be identified with a unique version identifier. The result is undefined if an implementation does not support an interchange file of the version that is defined in the version string.

```
<classDefinition> ::=
        'Class'  'named:'  <classNameString>
        'superclass:' <superclassNameString>
        'indexedInstanceVariables:' <indexableInstVarType>
        'instanceVariableNames:' <instanceVariableNames>
        'classVariableNames:' <classVariableList>
        'sharedPools:' <poolList>
        'classInstanceVariableNames:'<classInstVariableList>
        <elementSeparator>

<classNameString> ::= stringDelimiter <className> stringDelimiter
<superclassNameString> ::= stringDelimiter <className> stringDelimiter
<className> ::= identifier
<indexableInstVarType> ::= hashedString
<instanceVariableNames> ::= <identifierList>
<classVariableList> ::= <identifierList>
<classInstVariableList> ::= <identifierList>
<poolList> ::= <identifierList>
<identifierList> ::= stringDelimiter identifier* stringDelimiter

<methodDefinition> ::=
        <className> 'method' <elementSeparator>
        <method definition> <elementSeparator>

<classMethodDefinition> ::=
        <className> 'classMethod' <elementSeparator>
```

```
        <method definition> <elementSeparator>

<classInitializationr>  ::=
        <className> 'initializer' <elementSeparator>
        <initializer definition> <elementSeparator>
```

These productions correspond to the components of a <<class definition>> as defined in section 3.3.2. The <<instance state>> is defined in the interchange format by the <instanceVariableNames> and the <indexableInstVarType> clauses. If the hashedString of the <indexableInstVarType> clause is `#'byte'` then the <<instance state>> clause includes the <<*byte indexable*>> symbol. If the hashedString of the <indexableInstVarType> clause is `#'object'` then the <<instance state>> clause includes the <<*object indexable*>> symbol. If the hashedString of the <indexableInstVarType> clause is `#'none'` then the <<instance state>> clause includes neither the <<*byte indexable*>> symbol or the <<*object indexable*>> symbol. The <instanceVariablesNames> corresponds to <<instance variable names>>. The identifiers in this list correspond to the identifiers in <<class variable names>>. The identifiers are separated by whiteSpace.

The <classVariableList> corresponds to <<class variable names>>. The identifiers in this list correspond to the identifiers in <<class variable names>>. The identifiers are separated by whiteSpace.

<classInstVariableList> corresponds to <<class instance variable names>>. The identifiers in this string correspond to the identifiers in the <<class variable names>>. The identifiers are separated by whiteSpace.

<poolList> corresponds to <<imported pool names>>. The identifiers in this string correspond to the identifiers in the <<imported pool names>>. The identifiers are separated by whiteSpace.

The collection of all <methodDefinition> elements with a common class name throughout the file correspond to the <<instance methods>> portion of the <<class definition>> for that class name. The collection of all <classMethodDefinition> elements with a common class name through out the file correspond to the <<class methods>> portion of the <<class definition>>. Any <methodDefinition> elements or <classMethodDefinition> elements for a particular class name must follow the <classDefinition> for that class name.; however, they do not need to immediately follow the <classDefinition>. Nor do they need to be adjacent within the interchange file.

<classInitializationr> corresponds to the <<class initializer>> of a <<class definition>>. The <classInitialization> may appear anywhere following the <classDefinition> of the class to which it applies.

```
<globalDefinition> ::=
        <globalVariableDefinition>  | <globalConstantDefinition>

<globalVariableDefinition> ::=
        'Global' 'variable:' <globalNameString> <elementSeparator>
<globalConstantDefinition> ::=
        'Global' 'constant:' <globalNameString> <elementSeparator>
<globalValueInitialization> ::=
        <globalName> 'initializer' <elementSeparator>
        <variableInitializer> <elementSeparator>

<globalNameString> ::=
        stringDelimiter <globalName> stringDelimiter
<globalName> ::= identifier
<variableInitializer> ::= <initializer definition>
```

These productions provide the syntax of the interchange format for a <<global definition>>. The <globalConstantDefinition> corresponds to a <<global definition>> with a <<*constant designator*>> and the <globalVariableDefinition> corresponds to a <<global definition>> without a <<*constant designator*>>. The <globalName> corresponds to the <<global name>> of that <<global definition>>. The <variableInitializer> of a <globalValueInitialization> with an identical <globalName> corresponds to the <variableInitializer> of the corresponding <<global definition>>. The <globalValueInitialization> may appear anywhere following the <globalDefinition>.

```
<poolDefinition> ::=
        'Pool' 'named:' <poolNameString> <elementSeparator>
<poolVariableDefinition> ::=
        <poolValueDefinition> | <poolConstantDefinition>

<poolValueDefinition> ::= <poolName> 'variable:'
                        <poolVariableNameString> <elementSeparator>
<poolConstantDefinition> ::= <poolName> 'constant:'
                        <poolVariableNameString> <elementSeparator>
<poolValueInitialization> ::=
        <poolName> 'initializerFor:' <poolVariableNameString>
        <elementSeparator> <variableInitializer> <elementSeparator>
<poolNameString> ::=
        stringDelimiter <poolName> stringDelimiter
<poolVariableNameString> ::=
        stringDelimiter <poolVariableName> stringDelimiter
<poolName> ::= identifier
<poolVariableName> ::= identifier
```

These productions provide the syntax of the interchange format corresponding to a <<pool definition>>. A <poolName> corresponds to a <<pool name>> of the abstract program syntax. A <poolValueDefinition> defines a <<pool definition>> without a <<*constant designator*>> and a <poolConstantDefinition> defines a <<pool definition>> with a <<*constant designator*>>.

<poolVariableName> corresponds to a <<pool variable name>>. The individual <poolVariableDefinition> elements for a pool may appear anywhere following the <poolDefinition> for that pool. The <poolValueInitialization> elements may appear anywhere following the corresponding <poolValueDefinition>.

```
<programInitialization> ::=
        'Global' 'initializer' <elementSeparator>
        <programInitializer> <elementSeparator>

<programInitializer> ::= <initializer definition>
```

This is the interchange syntax for the <<program initializer definition>>. The <initializer definition> in the <programInitializer> corresponds to the <initializer definition> of a <<program initializer definition>>.

```
<annotation> ::=
        'Annotation' 'key:' quotedString
                'value:' quotedString <elementSeparator>
```

An <annotation> defines an implementation defined attribute for program element to which it is attached. These attributes have no semantics defined by this standard. They are provided as a mechanism for implementations to exchange extra-lingual information concerning the program. An

implementation is under no obligation to do anything with and can totally ignore any or all annotations.

Multiple annotations may follow a single definition or initialization. The first string of an <annotation> names the attribute defined by the annotation. The second string provides the value of that attribute.

For interchange purposes the following standard attributes are defined :

key	value
'category'	The name of a classification category of the language element
'comment'	A textual comment documenting the language element
'copyright'	The text of a copyright notice
'author'	Identifying information of the creator of the definition

Additional attributes may be defined by implementations. Implementors are encourage to choose attribute names that will not conflict with those chosen by other implementors. Implementors are also encouraged to cooperate in the naming of attributes that may be of general utility.

5. Standard Class Library

5.1 Definitions and Concepts

The class library specification defines the externally visible behavior of a set of concrete classes in a conforming Smalltalk system without supplying a corresponding implementation. The specification does not define how the classes must be arranged in an inheritance hierarchy, nor does it specify the existence or behavior of any abstract classes. Similarly, the specification does not specify the instance variables of classes in a conforming implementation.

In order to specify the standard class library without making reference to any particular inheritance hierarchy, the class library specification uses a behavioral description based on protocols. A *protocol* is a named semantic interface, defined by a *glossary of terms*, and a set of *message specifications*. Protocols are independent of implementation inheritance relationships, and are intended to clearly specify essential aspects of behavior while leaving incidental aspects unspecified. The fact that something is explicitly unspecified may be important information to both implementors and application developers.

Protocols are denoted by **<P>**, where P is the name of a protocol. Protocols are used in the specification both to define the behavior of concrete classes, and also to factor common behavior from other protocols through a relationship called *conformance*. Protocols that define the behavior of objects bound to global names (*concrete protocols*) start with an uppercase letter, and implementations must provide a global name that is the same as the name of the protocol which implements the specified behavior. Protocols that are used only as a factoring mechanism in the specification start with a lowercase letter and are referred to as *abstract protocols*.

For any particular set of concrete classes there are typically many possible factorings of behavior into abstract protocols. In general we will prefer the minimal set of abstract protocols which capture the concrete behavior being modeled, unless there are compelling reasons to the contrary, e.g.

certain operations have been traditionally grouped together. For greater clarity, our goal in defining the abstract protocols is to describe the behavior of the global names being specified in this standard. While the abstract protocols may also prove to be useful as a tool for specifying Smalltalk class libraries, it is not a goal to provide a set of generic, reusable abstract protocols for this purpose.

Rationale

Experience has shown that there is occasionally a conflict between adhering strictly to the protocol specification rules as laid out in this section, and providing a clear and unambiguous specification for a particular message. This arises most frequently in cases where the simplest and most obvious definition breaks the protocol conformance rules (covariance and contravariance). In all such cases we have opted for simplicity and clarity, even if the resulting specification does not strictly follow protocol conformance rules.

5.1.1 Glossary of Terms

The glossary of terms defines terminology that is used in the message specifications to describe message semantics. The glossary typically defines terms related to an abstract model of behavior for a particular object, and it is usually specific to the domain in which the object is applied. For example, one may consider that all collections behave as if they contain *elements*, regardless of whether elements actually exist in a particular implementation. The *concept* of elements is therefore useful in defining collection operations, and so it is defined in the glossary for the <collection> protocol. Note however, that the existence in the specification of a glossary term of a conceptual variable called *elements* does not require the implementation to actually have such an instance variable.

5.1.2 Message Specification

A message specification describes an individual message in the context of a particular protocol. It is defined by a *message selector*, a *behavioral description*, a set of *parameter specifications*, and a set of *return value specifications*. A specification for a particular message may appear in arbitrarily many protocols, and no two message specifications in the same protocol may have the same message selector. A message specification has no meaning outside of the context of a protocol.

A selector names a message, and is denoted as such with a preceding # symbol. For example, `#at:put:` is a 2-parameter message selector. We distinguish a message from a method as follows:

A selector, together with its parameters, is a *message*.

A selector, together with a receiver object, identifies a *method*, the unique implementation of the message.

Just as a method is uniquely identified by a receiver and selector, a message specification is uniquely identified by a protocol and selector pair (**<P>, s)** where **P** is a protocol name and **s** is a selector.

5.1.2.1 Behavioral Description

The behavior of a message is described with English text, using a definitional style wherever possible. Basic operations are described in terms of their effects on the abstract state of the object (using terms described in the glossary). These form the building blocks for specifying the behavior of more complex messages, which may be described in terms of the basic messages.

Words which are glossary entries are always in *italics*, and words which are formal parameter names are always in a `fixed-pitch font`. This eliminates confusion between a specific use of a word as defined in the glossary and normal English usage.

5.1.2.2 Parameter Specification

A parameter specification is defined by a parameter name, a parameter interface definition, and a parameter aliasing attribute.

A parameter specification places constraints on the parameter in terms of protocol conformance, and provides information concerning how the parameter is used by implementations of the message. The parameter name is the name of a formal parameter and is used to identify the parameter with a parameter specification, and to refer to the parameter in textual descriptions.

A parameter interface definition is defined as either:

- A single protocol name **<P>**.
- A logical OR of two or more protocols, written as **<P1> | <P2> | ... | <Pn>**

The parameter interface definition identifies the behavioral assumptions the message makes concerning the parameter. A client must supply an appropriate actual parameter. An OR of protocols means that the parameter must conform to at least one of the protocols in the disjunction. This is required to describe cases where a message accepts objects with diverse behavior and tests their behavior by sending messages in order to determine the action to be taken. Note that this is different from the case where a message accepts objects with diverse behavior, but only makes use of common shared behavior. In the latter case, the message is not really dealing with diverse cases of behavior.

When a message specifies that a given formal parameter must conform to a protocol **<P>**, it is making a commitment to use only behavior which is defined in **<P>** in the message implementation. In this sense, the conformance statement is a *maximal* behavioral requirement—at most all of the behavior described by **<P>** will be used, and no more.

Aliasing information (for example, whether a parameter is stored, or whether a returned value is new or returned state) is specified to avoid having implementors use defensive programming techniques which result in unnecessary object creation and copying, incurring a performance penalty. We differentiate between *incidental* aliasing and *essential* aliasing, both for parameters and for return values. Essential aliasing forms a critical part of the behavior of the interface, and as such it must be specified by the interface designer. Incidental aliasing should not be specified since it is a side effect of implementation choices, and is not fundamental to the specified functionality of the interface.

Essential aliasing of parameters is described using a parameter aliasing attribute:

captured	The receiver always retains a reference to the parameter, directly or indirectly, as a result of this message.
uncaptured	The receiver never retains a reference to the parameter, directly or indirectly, as a result of this message.
unspecified	It is unspecified as to whether or not a reference is retained as a result of this message i.e. either case may occur.

5.1.2.3 Return value specification

A *return value specification* is defined by a return value protocol and a return value aliasing attribute. Whereas the parameter description is *prescriptive* in that it states requirements to which the parameters must conform, the return value information is *descriptive* in that it provides information about the result being returned. Whereas a protocol makes a conformance *requirement* statement about parameters, it makes a conformance *commitment* concerning the return value. The specification guarantees that the return value will conform to the specified protocol.

A message specification may have multiple distinct return value specifications. Conversely, a single return value specification may describe multiple return values if the return value specification

applies to all such values. Multiple return value specifications are required for cases where a message is defined to return objects conforming to different protocols, on a case-specific basis. These are conveniently described with separate conformance statements and aliasing annotations. In order to establish correspondence between sets of return value specifications, we do not permit two distinct return value specifications which promise conformance to the same protocol.

If a message specification has no return value specification (that is, the return value is not specified), then it is not prepared to guarantee anything about the behavior of the returned object. In this case we denote the return value as **UNSPECIFIED**. This can be used to separate procedural messages from functional messages; to allow for inconsequential differences in implementations; or to allow conforming implementations which return different results but are otherwise operationally equivalent.

In order to relate return values through conformance, we define the return value interface definition for a message specification to be the single return value protocol, or the logical OR of the protocols in each distinct return value specification.

Information concerning retained references to return values (by the message receiver) is described using a return value aliasing attribute, which is one of the following identifiers:

state	The receiver retains a reference (direct or indirect) to the returned object after the method returns i.e. the object is returned state.
new	The object is newly created in the method invocation and no reference (direct or indirect) is retained by the receiver after the method returns.
unspecified	No information is provided as to the origin or retained references to the object (Note this is different from saying that the return value itself is **UNSPECIFIED**. Here we are committing that the return value conforms to some protocol, but making no commitment about the aliasing behavior).

Note that we do not attempt to describe the aliasing of the state variables of the return value itself—the attribute applies only to the first level returned object. The implication is that second and subsequent level aliasing of the return value is *always* unspecified. An exception occurs in the case where the returned state is an object which the client originally gave the service provider for safekeeping. This occurs with element retrieval in collections, for example. In such cases only the client knows the implications of modifying second level state of the return value.

5.1.3 Conformance and Refinement

Protocols are related to each other through two substitutability relationships, *conformance* and *refinement,* which arrange the protocols in a lattice. Conformance models requirements satisfaction, and provides the flexibility to partially constrain the behavior of parameters and return values without necessarily naming specific classes. Refinement allows a protocol to make more precise statements about behavior inherited from another protocol.

5.1.3.1 Conformance

Conformance can be defined on both objects and protocols.

5.1.3.1.1 Object Conformance

An object **x** conforms to a protocol **<P>** if it implements the set of behaviors specified by **<P>**. Of course, such an object may have additional behavior, or it may be possible to specify the object's behavior in more detail. Thus a protocol may describe only a subset of an object's behavior, or it may leave certain aspects of the behavior unspecified.

We require that all objects of the same class necessarily conform to the same protocols, since they have the same implementation. This is the same relationship between classes and behavior

established by most object-oriented type systems and allows us to establish a straightforward relationship between classes and protocols.

5.1.3.1.2 Protocol Conformance

Protocols are also related to each other through conformance. If all objects that conform to a protocol **<P>** also conform to a protocol **<Q>**, then **<P>** is defined to conform to **<Q>**. If both **<P>** conforms to **<Q>** and **<Q>** conforms to **<P>** then necessarily **<P>** and **<Q>** define the same behavior i.e. **<Q> = <P>**.

A protocol **<P>** conforms to a protocol **<Q>** if and only if both of the following are true:

- Every message specification in **<Q>** has a corresponding message specification in **<P>** with the same message selector.
- Every message specification in **<P>** conforms to its corresponding message specification in **<Q>**.

This means that in order for **<P>** to conform to **<Q>**, **<P>** must define message specifications for at least all of the message selectors in **<Q>**, and these specifications must specify compatible behavior.

Note that conformance is a transitive relationship. If an object or a protocol conforms to a protocol through transitivity we say that it *implicitly* conforms.

5.1.3.1.3 Message Specification Conformance

In the context of protocols, message specifications are related through conformance. A message specification **s** in a protocol **<P>** is identified by a protocol and selector pair (**<P>, s)**. A message specification (**<P>, s)** conforms to a message specification (**<Q>,s)** if and only if all of the following are true:

1. **(<P>, s)** and (**<Q>, s)** have the same formal parameters (the same names in the same positions).
2. the parameter interface definitions of (**<Q>, s)** conform to the corresponding parameter interface definitions of (**<P>, s)** (*contravariance*).
3. the parameter aliasing attributes of (**<P>, s**) conform to the corresponding parameter aliasing attributes of (**<Q>, s)** (*covariance*).
4. the return value interface definition of (**<P>, s)** conforms to the return value interface definition of (**<Q>, s)** (*covariance*).
5. the return value aliasing attributes of (**<P>, s)** conforms to the corresponding return value aliasing attributes of (**<Q>, s)** (*covariance*).
6. the behavioral description of (**<P>, s)** conforms to the behavioral description of (**<Q>, s)** (*covariance*).

5.1.3.1.4 Interface Definition Conformance

Recall that a parameter or return value interface definition is either a single protocol, or a logical OR of two or more protocols. The protocol set for an interface definition is the set of protocols in the disjunction (or the set consisting of a single protocol). An interface definition **I** conforms to an interface definition **J** if and only if the protocol set for **I** is a subset of the protocol set for **J**. (Note that we do not require a proper subset; the sets may be equal.)

Note that interface definition conformance is defined by a subset relationship, since an interface definition is defined to require an object conforming to one or more protocols in the corresponding disjunction. The subset relationship follows directly from this definition.

5.1.3.1.5 Parameter Aliasing Conformance

Parameter aliasing attributes that are the same conform to each other. The following additional conformance relationships are also defined among the parameter aliasing attributes:

captured conforms to **unspecified**.
uncaptured conforms to **unspecified**.

5.1.3.1.6 Return Value Aliasing Attribute Conformance

Return value aliasing attributes that are the same conform to each other. The following additional conformance relationships are also defined among the return value aliasing attributes:

state conforms to **unspecified**.
new conforms to **unspecified**.

5.1.3.1.7 Behavioral Description Conformance

A behavioral description **D2** conforms to a behavioral description **D1**, if the behavior described by **D1** is implied by **D2**. If **D2** actually includes the text of **D1**, it is more difficult for a designer to accidentally violate conformance in a message specification when conformance between protocols has been asserted. In most cases, contradictions are readily apparent. Consequently this is the recommended practice.

The conformance rules for behavioral descriptions reflect the fact that substitutability requires that behavior be strictly additive. Conforming protocols may only define new messages, or provide more precise statements concerning the behavior of existing messages.

5.1.3.2 Refinement

The refinement relation can be applied wherever we have defined the conformance relation, and is defined as follows. Given any **A** and **B** such that there is a conformance relation defined on **A** and **B**, then **A** is a refinement of **B** if **A** conforms to **B** but **B** does not conform to **A**. Refinement applies to protocols, message specifications, interface definitions, and aliasing attributes. It is also convenient to say that if **A** is a refinement of **B**, then **A** refines **B**.

Refinement makes a stronger statement than conformance. Refinement describes the property which relates protocols in a way that allows them to make progressively more and more precise statements concerning object behavior, while still satisfying conformance. As a consequence, given two protocols **<A>** and **<B>** such that **<A>** refines **<B>**, objects which conform to **<A>** are substitutable for objects which conform to **<B>**.

We call a message specification in a protocol **<P>** a *definition* if it is not included in any protocol to which **<P>** conforms. Otherwise it is a *refinement*. The protocol conformance relationship defines a lattice with protocols at the nodes. For any given message specification (**<P>, s)** in a protocol **<P>** there exists a path through refinements (if any) to the definition (**<P'>, s')** of the message specification. The set of all paths from **<P>** establishes message specification visibility. A message specification (**<Q>, t)** is visible from a protocol **<P>** if either **<P> = <Q>** or there exists a path from **<P>** to **<Q>**. The *implicit specification* of a message in a protocol **<P>** is the closest message specification visible from **<P>** through the conformance graph (i.e., involving the fewest number of arcs in the graph).

Thus the implicit specification may be either in **<P>** or in some protocol to which **<P>** conforms. Note that we use the closest visible message specification in order to ensure we obtain all

refinements in the refinement path. Since protocol conformance forms a directed acyclic graph, there can in principle be multiple conformance paths to the same message specification. In such a case we explicitly disallow conflicts. If there are multiple implicit message specifications for the same message selector found by traversing different paths, they must result in the same specification.

The full text of the behavioral description of a message specification is obtained by concatenating the behavioral description from the definition together with the text added by behavioral refinements in the visibility path, in reverse order.

In summary, a message specification may form a part of the behavior described by a protocol **<P>** in the following ways:

Definition	**<P>** contains the definition.
Refinement	**<P>** contains a refinement of an implicit specification of the message.
Conformance	an implicit message specification is visible and there is no refinement in **<P>**.

5.1.3.3 Special Protocols

Two special protocols are defined:

<ANY>	A protocol to which all other protocols conform.
<RECEIVER>	A notational convenience which represents the protocol to which the receiver of the message conforms.

All objects are defined to conform to the special protocol **<ANY>**. The protocol **<ANY>** places no restrictions on a parameter definition since it allows all possible parameters; **<ANY>** may be thought of as a protocol which specifies no behavior.

The **<RECEIVER>** protocol is a notational convenience that allows a message specification to indicate a return value which conforms to the protocol in which it is used, or any protocol that conforms to that protocol. Due to the contravariance requirement for parameter interface definition conformance, **<RECEIVER>** cannot be used in a parameter specification since it is necessarily covariant. However, for the same reason, it is valid in a return value specification.

5.1.4 Protocol Specification Conventions

5.1.4.1 Naming

A protocol's name has its initial letter capitalized if there is a global name defined in the standard that is conformant to the protocol. For instance, <OrderedCollection> has its first letter capitalized but <puttableStream> does not.

Protocols that are required to be implemented as class objects in Smalltalk implementations end with the word "class". Protocols that are typically implemented as class objects, but are not required to be so, end with either the word "factory", if they are used to create new objects, or the word "discriminator".

5.1.4.2 Message Lists

Each protocol includes a list of the message selectors defined or refined by the protocol. If a message is refined by the protocol it is shown in italics in this list.

5.1.4.3 Message Definitions

Message definitions have these elements:

- A header that shows the message pattern. The message pattern is preceded by the word "**Message:**" or for refinements of messages defined in other protocols, "**Message Refinement:**".
- A synopsis, which is a short and informal description of what the message does, under the heading "**Synopsis**".
- A more rigorous definition of the message. The heading for this section, "**Definition:**", is followed by the name of the defining protocol. For refinements, the text of the inherited definition is merely copied.
- For each inherited refinement and the current protocol's refinement, a refinement section showing how the method is refined. The heading for this section, "**Refinement:**", is followed by the name of the refining protocol.
- A list of the parameters of the message under the heading "**Parameters**", what their required protocol conformance is, and whether they are captured by the receiver of the message. Each parameter is listed on a separate line using the format:

  parameterName <parametersProtocol> captured/uncaptured/unspecified

  If there are no parameters, this element is omitted.
- A description of the return value, under the heading "**Return Value**", in the form:

  <returnValueProtocol> state/new/unspecified

  or

  UNSPECIFIED
- A list of errors that define erroneous conditions for the message under the heading "**Errors**".

For example,

Message: canAcceptSalaryIncrease: amount

Synopsis

Determine whether the receiver can accept the given salary increase.

Definition: <Employee>

This message determines whether the receiver is allowed to receive the given salary increase.

It answers `true` if the elevated salary is acceptable and `false` if not.

Parameters

`amount`	<scaledDecimal>	uncaptured

Return Value

<boolean> unspecified

Errors

none

or,

Message Refinement: canAcceptSalaryIncrease: amount

Synopsis

Determine whether the receiver can accept the given salary increase.

Definition: <Person>

This message determines whether the receiver is allowed to receive the given salary increase.

It answers `true` if the elevated salary is acceptable and `false` if not.

Refinement: <Employee>

This refines the inherited message by checking the amount against known consistency rules for an employee object.

Parameters

`amount`	<scaledDecimal>	uncaptured

Return Value

<boolean> unspecified

Errors

none

In the second example, the message is a refinement of the definition from protocol <Person> and is refined in <Employee>.

5.1.4.4 Protocol Groupings

Within a grouping, protocols are ordered according to their conformance lattice. Secondary sorting is alphabetical by protocol name.

5.2 Standard Globals

The following global values exist with the named protocols in Standard-conforming implementations. The values of the globals are objects that conform to the specified protocols. The language element type identifies the type of Smalltalk language element identified by the global. Valid language elements are Class, Global Variable, Named Object (a constant global), or Pool. An implementation may implement a global with an "unspecified" language element type as any of these element types except as a Pool.

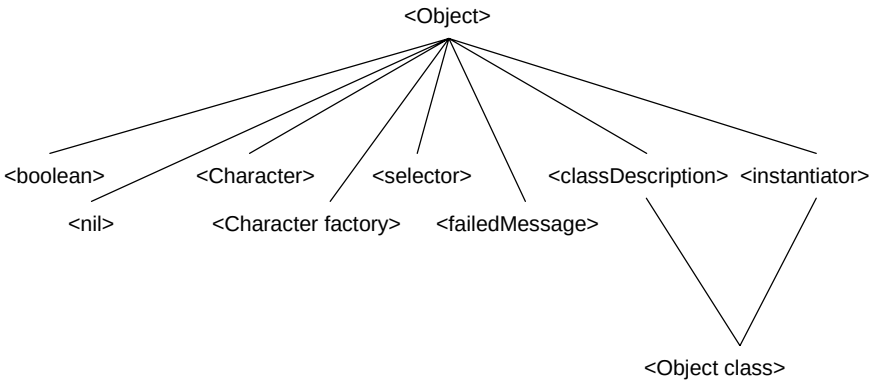
Name of global	Protocol	Language Element	Grouping
Array	<Array factory>	unspecified	Collection Protocols
Bag	<Bag factory>	unspecified	Collection Protocols
ByteArray	<ByteArray factory>	unspecified	Collection Protocols
DateAndTime	<DateAndTime factory>	unspecified	DateTime Protocols
Dictionary	<Dictionary factory>	unspecified	Collection Protocols
Duration	<Duration factory>	unspecified	DateTime Protocols
Error	<Error class>	Class	Exception Protocols
Exception	<Exception class>	Class	Exception Protocols
FileStream	<FileStream factory>	unspecified	File I/O Protocols
Float	<floatCharacterization>	unspecified	Numeric Protocols
FloatD	<floatCharacterization>	unspecified	Numeric Protocols
FloatE	<floatCharacterization>	unspecified	Numeric Protocols
FloatQ	<floatCharacterization>	unspecified	Numeric Protocols
Fraction	<Fraction factory>	unspecified	Numeric Protocols
IdentityDictionary	<Identitydictionary factory>	unspecified	Collection Protocols
Interval	<Interval factory>	unspecified	Collection Protocols
MessageNotUnderstood	<messageNotUnderstoodSelector>	unspecified	Exception Protocols
Notification	<Notification class>	Class	Exception Protocols
Object	<Object class>	Class	Fundamental Protocols
OrderedCollection	<OrderedCollection factory>	unspecified	Collection Protocols
ReadStream	<ReadStream factory>	unspecified	Stream Protocols
ReadWriteStream	<ReadWriteStream factory>	unspecified	Stream Protocols
Set	<Set factory>	unspecified	Collection Protocols
SortedCollection	<SortedCollection factory>	unspecified	Collection Protocols

String	<String factory>	unspecified	Collection Protocols
TimeLocal	<TimeLocal factory>	unspecified	DateTime Protocols
Transcript	<Transcript>	unspecified	Stream Protocols
Warning	<Warning class>	Class	Exception Protocols
WriteStream	<WriteStream factory>	unspecified	Stream Protocols
ZeroDivide	<ZeroDivide factory>	Class	Exception Protocols

5.3 Fundamental Protocols

This section includes protocols that are fundamental to the Smalltalk language.

The graph below shows the conformance relationships between the protocols defined in this section.



5.3.1 Protocol: <Object>

Conforms To

<ANY>

Description

This protocol describe the behavior that is common to all objects.

Standard Globals

`Integer`	Conforms to the protocol <Object>. Its language element type is unspecified. This global identifies integer objects.
`Number`	Conforms to the protocol <Object>. Its language element type is unspecified. This global identifies number objects.
`ScaledDecimal`	Conforms to the protocol <Object>. Its language element type is unspecified. This global identifies scaled decimal objects.
`Symbol`	Conforms to the protocol <Object>. Its language element type is unspecified. This global identifies objects that conform to the protocol <symbol>.

Messages

=
==
~=
~~
class
copy
doesNotUnderstand:
error:
hash
identityHash
isKindOf:
isMemberOf:
isNil
notNil
perform:
perform:with:
perform:with:with:
perform:with:with:with:
perform:withArguments:
printOn:
printString
respondsTo:
yourself

Rationale

Several groups of messages that might be expected to be found here have not been included in the specification. The reasons for each are discussed below.

Dependents protocols were not included because there is nothing defined by the standard that requires any kind of dependency mechanism.

The messages `#storeOn:` and `#storeString` were excluded for the following reason: Creating a Smalltalk expression that can reconstruct an object is only useful in a application if there is a mechanism for an application to take such a string and compile and execute it. Because the ability to perform runtime compilation is something we do not want to mandate #storeOn: will not be useful in a compliant program. An alternative would be to provide messages to externalize and internalize object structures without specify a particular externalization representation. This would enable portable programs that externally store objects. However it would not enable interchange of objects between different implementations.

Design description messages such as #subclassResponsibility, #shouldNotImplement, #implementedBySubclass have not been included in this protocol. The rationale is that they are design documentation aids, not true execution behavior of objects. As such they should be supported either by explicit language extensions or the development environment .

5.3.1.1 Message: = comparand

Synopsis

Object equivalence test.

Definition: <Object>

This message tests whether the receiver and the `comparand` are equivalent objects at the time the message is processed. Return *true* if the receiver is equivalent to `comparand`. Otherwise return *false*.

The meaning of "equivalent" cannot be precisely defined but the intent is that two objects are considered equivalent if they can be used interchangeably. Conforming protocols may choose to more precisely define the meaning of "equivalent".

The value of

```
receiver = comparand
```

is *true* if and only if the value of

```
comparand = receiver
```

would also be *true*. If the value of

```
receiver = comparand
```

is *true* then the receiver and `comparand` must have *equivalent hash values*. Or more formally:

```
receiver = comparand
receiver hash = comparand hash
```

The equivalence of objects need not be *temporally invariant*. Two independent invocations of #= with the same receiver and `operand` objects may not always yield the same results. Note that a collection that uses #= to discriminate objects may only reliably store objects whose hash values do not change while the objects are contained in the collection.

Parameters

`comparand`	<Object>	uncaptured

Return Value

<boolean>	unspecified

Errors

none

Rationale

Note that object equality is not explicitly defined as being the same as object identity. That is probably the only reasonable implementation in Object but not specifying it allows this protocol to be used without refinement by other classes with more precise definitions of equivalence

5.3.1.2 Message: == comparand

Synopsis

Object identity test.

Definition: <Object>

This message tests whether the receiver and the `comparand` are the same object. Return *true* if the receiver is the same object as `comparand`. Otherwise return *false*.

The value of

```
receiver == comparand
```

is *true* if and only if the value of

```
comparand == receiver
```

would also be *true*. If the value of

```
receiver == comparand
```

is *true* then the receiver and `comparand` must have *equivalent identity hash value*s. Or more formally:

```
receiver == comparand
receiver identityHash = comparand identityHash
```

Parameters

`comparand`	<Object>	uncaptured

Return Value

<boolean>	unspecified

Errors

none

5.3.1.3 Message: ~= comparand

Synopsis

Object inequality test.

Definition: <Object>

This message tests whether the receiver and the `comparand` are not *equivalent* objects at the time the message is processed. Return *true* if the receiver is not *equivalent* to `comparand`. Otherwise return *false*.

The meaning of "equivalent" cannot be precisely defined but the intent is that two objects are considered equivalent if they can be used interchangeably. Conforming protocols may choose to more precisely define the meaning of "equivalent".

The result must be *equivalent* to the Boolean negation of the result of sending the message #= to the receiver with `comparand` as the argument.

The value of

```
receiver ~= comparand
```

is *true* if and only if the value of

```
comparand ~= receiver
```

would also be *true*.

Parameters

`comparand`	<Object>	uncaptured

Return Value

<boolean>	unspecified

Errors

none

Rationale

This definition does not require that the implementation be:
^(self = comparand) not
but it does require that within a class, the same definition of equivalence is used in the implementation of both #= and #~=.

5.3.1.4 Message: ~~ comparand

Synopsis

Negated object identity test.

Definition: <Object>

This message tests whether the receiver and the `comparand` are different objects. Return *true* if the receiver is not the same object as `comparand`. Otherwise return *false*.

The result must be equivalent to the Boolean negation of the result of sending the message #== to the receiver with `comparand` as the argument.

The value of

```
receiver ~~ comparand
```

is *true* if and only if the value of

```
comparand ~~ receiver
```

would also be *true*.

Parameters

`comparand`	<Object>	uncaptured

Return Value

<boolean>	unspecified

Errors

none

Rationale

This definition does not require that the implementation be:
^(self == comparand) not
but it does require that the same definition of object identity is used in the implementation of both #== and #~~.

5.3.1.5 Message: class

Synopsis

Determine the class of the receiver.

Definition: <Object>

If the receiver is an instance object, return the *class object* defined by the *class definition* that defines the behavior of the receiver. If the receiver is itself a *class object*, the result is unspecified except that it must conform to the protocol <classDescription>.

Return Value

<classDescription>	unspecified

Errors

none

5.3.1.6 Message: copy

Synopsis

Return a copy of the receiver.

Definition: <Object>

Return a new object that must be as similar as possible to the receiver in its initial state and behavior. Any operation that changes the state of the new object should not as a side-effect change the state or behavior of the receiver. Similarly, any change to the receiver should not as a side-effect change the new object.

If the receiver is an *identity object*, return the receiver.

Return Value

<RECEIVER> unspecified

Errors

none

Rationale

An argument can be made that the receiver and the result should respond *true* to an #= test. However, the traditional definition of Object #= in terms of #== is incompatible with this requirement.

5.3.1.7 Message: doesNotUnderstand: message

Synopsis

A message was sent to the receiver for which the receiver has no behavior.

Definition: <Object>

A message was sent to the receiver for which the receiver has no behavior. Signal a MessageNotUnderstood exception corresponding to the failed message. If the exception resumes, the resumption value is returned as the value of this message.

Conforming protocols may refine this message to perform some action other than signaling the exception.

Parameters

`message` <failedMessage> unspecified

Return Value

<Object> unspecified

Errors

none

5.3.1.8 Message: error: signalerText

Synopsis

Announce an error

Definition: <Object>

This message is used the announce the occurrence of some type of error condition. The argument should describe the nature of the error. The default behavior is to raise an Error exception as if the message `#signal:` had been sent to the global Error with `signalerText` as the argument.

Conforming protocols may refine this message to perform some action other than signaling the exception.

Parameters

`signalerText` <readableString> unspecified

Return Value

UNSPECIFIED

Errors

none

5.3.1.9 Message: hash

Synopsis

Return an integer hash code that can be used in conjunction with an #= comparison.

Definition: <Object>

An integer value that can be used as a hash code for the receiver is returned. The hash code is intended for use in conjunction with an #= comparison.

The range, minimum, and maximum values of the result is implementation defined.

Any two objects that are considered *equivalent* using the #= message must have the same hash value. More formally:

```
receiver = comparand
receiver hash = comparand hash
```

The *hash value* of an object need not be *temporally invariant*. Two independent invocations of `#hash` with the same receiver may not always yield the same results. Note that collections that use #= to discriminate objects may only reliably store objects whose hash values do not change while the objects are contained in the collection.

Return Value

<integer>unspecified

Errors

none

5.3.1.10 Message: identityHash

Synopsis

Return an integer hash code that can be used in conjunction with an #== (identity) comparison.

Definition: <Object>

An integer value that can be used as a hash code for the receiver is returned. The hash code is intended for use in conjunction with an #== comparison.

The range, minimum, or maximum values of the result is implementation defined.

The identity hash of an object must be *temporally invariant*.

Return Value

<integer>unspecified

Errors

none

Rationale

Some existing implementations use the selector `#basicHash` for this message. `#basicHash` is inappropriate because of the convention that selectors starting with the sequence "basic" are private to the implementation of an object.

5.3.1.11 Message: isKindOf: candidateClass

Synopsis

Classify an object.

Definition: <Object>

Return *true* if the receiver is an instance of `candidateClass` or is an instance of a *general subclass* of `candidateClass`. Otherwise return *false*.

The return value is unspecified if the receiver is a *class object* or `candidateClass` is not a *class object*.

Parameters

`candidateClass` <Object> uncaptured

Return Value

<boolean> unspecified

Errors

none

5.3.1.12 Message: isMemberOf: candidateClass

Synopsis

Determine whether the receiver is an instance of the argument.

Definition: <Object>

Return *true* if the receiver is an instance of `candidateClass`. Otherwise return *false*.

The return value is unspecified if the receiver is a *class object* or `candidateClass` is not a *class object*.

Parameters

`candidateClass` <Object> uncaptured

Return Value

<boolean> unspecified

Errors

none

5.3.1.13 Message: isNil

Synopsis

Determine if the receiver is the value of the reserved identifier *nil*.

Definition: <Object>

Determine if the receiver is the same object as the value of the reserved identifier *nil*. Return *true* if it is, *false* if it is not.

The messages `#isNil` and `#notNil` must be implemented to produce consistent results. For a given receiver if the result of `#isNil` is *true* then the result of `#notNil` must be *false*.

Return Value

<boolean> unspecified

Errors

none

5.3.1.14 Message: notNil

Synopsis

Determine if the receiver is not the value of the reserved identifier *nil*.

Definition: <Object>

Determine if the receiver is the same object as the value of the reserved identifier *nil*. Return *false* if it is, *true* if it is not.

The messages `#isNil` and `#notNil` must be implemented to produce consistent results. For a given receiver if the result of `#isNil` is *true* then the result of `#notNil` must be *false*.

Return Value

<boolean> unspecified

Errors

none

5.3.1.15 Message: perform: selector
Message: perform: selector with: argument1
Message: perform: selector with: argument2
Message: perform: selector with: argument1 with: argument2
Message: perform: selector with: argument1 with: argument2 with: argument3

Synopsis

Send a message using a computed message selector.

Definition: <Object>

Send to the receiver a message whose selector is `selector` and whose arguments are `argument1`, `argument2`, etc. Return the value of that message.

If the receiver does not have a method for `selector` normal "message not understood" processing is performed as if the computed message hand been sent using a message send expression. If this occurs, `selector and the arguments` may be captured.

The perform messages and `#respondsTo:` must be implemented to produce consistent results. A message to perform a selector, `selector`, for a given receiver will result in a "message not understood" condition if and only if the value of

```
receiver respondsTo: selector
```

is *false*.

Behavior is undefined if the number of arguments does not match that implicitly required by the syntactic form of the `selector`.

Parameters

`selector`	<selector>	unspecified
`argument1`	<ANY>	unspecified
`argument2`	<ANY>	unspecified
`argument3`	<ANY>	unspecified

Return Value

<ANY> unspecified

The protocol specification of the returned value of this method is not really useful for any sort of static analysis. In practice the returned value will be treated as conforming to the return type of the message that is dynamically constructed.

Errors

none

5.3.1.16 Message: perform: selector withArguments: arguments

Synopsis

Send a message using a computed message selector and a collection of arguments.

Definition: <Object>

Send to the receiver a message whose selector is `selector` and whose arguments are the elements of `arguments`. Return the value of that message. The first element of `arguments` is the first argument, the second element is the second argument, and so on.

If the receiver does not have a method for the `selector` normal "message not understood" processing is performed as if the computed message hand been sent using a message send expression. If this occurs, `selector` and `arguments` could be captured.

The perform messages and `#respondsTo:` must be implemented to produce consistent results. A message to perform a selector, `selector`, for a given receiver will result in a "message not understood" condition if and only if the value of

```
receiver respondsTo: selector
```

is *false*.

Behavior is undefined if the number of elements in `arguments` does not match that implicitly required by the syntactic form of the `selector`.

Parameters

`selector`	<selector>	unspecified
`arguments`	<Array>	unspecified

Return Value

<ANY> unspecified

The protocol specification of the returned value of this method is not really useful for any sort of static analysis. In practice the returned value will be treated as conforming to the return type of the message that is dynamically constructed.

Errors

none

5.3.1.17 Message: printOn: target

Synopsis

Write a textual description of the receiver to a stream.

Definition: <Object>

The string of characters that would be the result of sending the message `#printString` to the receiver is written to `target`. The characters appear on the stream as if each character was, in sequence, written to the stream using the message `#nextPut:`.

Parameters

`target`	<puttableStream>	uncaptured

Return Value

UNSPECIFIED

Errors

none

5.3.1.18 Message: printString

Synopsis

Return a string that describes the receiver.

Definition: <Object>

A string consisting of a sequence of characters that describe the receiver are returned as the result.

The exact sequence of characters that describe an object are implementation defined.

Return Value

<readableString> unspecified

Errors

none

5.3.1.19 Message: respondsTo: selector

Synopsis

Determine if the receiver can respond to a specific message selector.

Definition: <Object>

Return *true* if the receiver has a method in its behavior that has the *message selector* `selector`. Otherwise return *false*.

Parameters

`selector`	<selector>	uncaptured

Return Value

<boolean>	unspecified

Errors

none

Rationale

Requiring this message should not significantly encumber implementations because that data structures and algorithms necessary to implement it at run time are essentially the same that are required to implement normal message lookup processing.

5.3.1.20 Message: yourself

Synopsis

No operation. Return the receiver as the result.

Definition: <Object>

Return the receiver of the message.

Return Value

<RECEIVER>	unspecified

Errors

none

5.3.2 Protocol: <nil>

Conforms To

<Object>

Description

This protocol describes the behavior that is unique to the distinguished immutable, *identity object* that is the value of the reserved identifier "nil".

Messages

printString

5.3.2.1 Message Refinement: printString

Synopsis

Return a string that describes the receiver.

Definition: <Object>

A string consisting of a sequence of characters that describe the receiver are returned as the result.

The exact sequence of characters that describe an object are implementation defined.

Refinement: <nil>

Return a string with the same characters as the string `'nil'`.

Return Value

<readableString> unspecified

Errors

none

5.3.3 Protocol: <boolean>

Conforms To

<Object>

Description

This protocol describes the behavior of the objects that are the values of the reserved identifiers "true" and "false". These objects are *identity objects*.

Several message specifications include a truth table describing the result of the binary operation implemented by that message. In each table, the value of the receiver is used to locate a row and the value of the argument is used to locate a column, the result being located at the intersection of the row and column.

Messages

&
|
and:
eqv:
ifFalse:
ifFalse:ifTrue:
ifTrue:
ifTrue:ifFalse:
not
or:
printString
xor:

5.3.3.1 Message: & operand

Synopsis

Logical and — Boolean conjunction.

Definition: <boolean>

Return the Boolean conjunction of the receiver and `operand`. The value returned is determined by the following truth table:

&	true	false

true	true	false
false	false	false

Parameters

`operand` <boolean> uncaptured

Return Value

<boolean> unspecified

Errors

none

5.3.3.2 Message: | operand

Synopsis

Logical or — Boolean disjunction.

Definition: <boolean>

Return the Boolean disjunction of the receiver and `operand`. The value returned is determined by the following truth table:

|	true	false
true	true	true
false	true	false

Parameters

`operand` <boolean> uncaptured

Return Value

<boolean> unspecified

Errors

none

5.3.3.3 Message: and: operand

Synopsis

"Short circuit" logical and.

Definition: <boolean>

If the receiver is *false*, return *false*. Otherwise, return the <boolean> result of sending the message `#value` to `operand`.

The result is undefined if the result of sending `#value` to `operand` is not a <boolean>.

Rationale

Some existing implementations do not require that the `operand` must evaluate to a <boolean>. The message `#ifTrue:` should be used to conditionally evaluate a block that does not return a <boolean>.

Parameters

`operand` <niladicBlock> uncaptured

Return Value

<boolean> unspecified

Errors

none

5.3.3.4 Message: eqv: operand

Synopsis

Boolean equivalence.

Definition: <boolean>

Return the Boolean disjunction of the receiver and `operand`. The value returned is determined by the following truth table:

eqv:	true	false
true	true	false
false	false	true

Parameters

`operand` <boolean> uncaptured

Return Value

<boolean> unspecified

Errors

none

5.3.3.5 Message: ifFalse: operand

Synopsis

Evaluate the argument if receiver is *false*.

Definition: <boolean>

If the receiver is *false* return the result of sending the message `#value` to `operand`.

The return value is unspecified if the receiver is *true*.

Rationale

Most existing implementations define the return value to be *nil* if the receiver is *true*. This definition is less precise and potentially allows for implementation specific optimization.

Parameters

`operand` <niladicBlock> uncaptured

Return Value

<ANY> unspecified

Errors

none

5.3.3.6 Message: ifFalse: falseOperand ifTrue: trueOperand

Synopsis

Selectively evaluate one of the arguments.

Definition: <boolean>

If the receiver is *false* return the result return the result as if the message `#value` was sent to `falseOperand`, otherwise return the result as if the message `#value` was sent to `trueOperand`.

Parameters

`falseOperand` <niladicBlock> uncaptured

`trueOperand` <niladicBlock> uncaptured

Return Value

<ANY> unspecified

Errors

none

5.3.3.7 Message: ifTrue: operand

Synopsis

Evaluate the argument if the receiver is *true*.

Definition: <boolean>

If the receiver is *true*, return the result of sending the message `#value` to `operand`.

The return value is unspecified if the receiver is *false*.

Rationale

Most existing implementations define the return value to be *nil* if the receiver is *false*. This definition is less precise and potentially allows for implementation specific optimization.

Parameters

`operand`	<niladicBlock>	uncaptured

Return Value

<ANY>	unspecified

Errors

none

5.3.3.8 Message: ifTrue: trueOperand ifFalse: falseOperand

Synopsis

Selectively evaluate one of the arguments.

Definition: <boolean>

If the receiver is *true* return the result of sending the message `#value` to `trueOperand`, otherwise return the result of sending `#value` to the `falseOperand`.

Parameters

`trueOperand`	<niladicBlock>	uncaptured
`falseOperand`	<niladicBlock>	uncaptured

Return Value

<ANY>	unspecified

Errors

none

5.3.3.9 Message: not

Synopsis

Logical not — Boolean negation.

Definition: <boolean>

Return the Boolean negation of the receiver.

If the receiver is *true* the return value is *false*, if the receiver is *false* the return value is *true*.

Return Value

<boolean>	unspecified

Errors

none

5.3.3.10 Message: or: operand

Synopsis

"Short circuit" logical or.

Definition: <boolean>

If the receiver is *true*, return *true*. Otherwise, return the Boolean result of sending the message `#value` to `operand`.

The result is undefined if the result of sending `#value` to `operand` is not a <boolean>.

Rationale

Some existing implementations do not require that the `operand` must evaluate to a <boolean>. The message `#ifFalse:` should be used to conditionally evaluate a block that does not return a Boolean.

Parameters

`operand` <niladicValuable> uncaptured

Return Value

<boolean> unspecified

UNSPECIFIED

Errors

none

5.3.3.11 Message Refinement: printString

Synopsis

Return a string that describes the receiver.

Definition: <Object>

A string consisting of a sequence of characters that describe the receiver are returned as the result.

The exact sequence of characters that describe an object are implementation defined.

Refinement: <boolean>

If the receiver is *true*, return a string with the same characters as the string `'true'`, otherwise return a string with the same characters as the string `'false'`.

Return Value

<readableString> unspecified

Errors

none

5.3.3.12 Message: xor: operand

Synopsis

Boolean exclusive or.

Definition: <boolean>

Return the Boolean exclusive or of the receiver and `operand`. The value returned is determined by the following truth table:

xor:	true	false
true	false	true
false	true	false

Parameters

`operand` <boolean> uncaptured

Return Value

<boolean> unspecified

Errors

none

5.3.4 Protocol: <Character>

Conforms To

<Object>

Description

This protocol describes the behavior that is common to character objects. Character objects serve as the element value for Smalltalk strings. The Smalltalk language provides a literal syntax for character objects. Character objects represent individual elements of an implementation defined execution character set whose individual elements are identified by integer values. These integers are called *code points*. Each character object has an associated code point.

It is unspecified whether or not each code point is uniquely associated with a unique character object.

The execution character set is the character set used by an implementation during execution of a Smalltalk program. It need not be the same as the character set used by that implementation to encode the definition of Smalltalk programs.

Messages

=
asLowercase
asString
asUppercase
codePoint
isAlphaNumeric
isDigit
isLetter
isLowercase
isUppercase

5.3.4.1 Message Refinement: = comparand

Synopsis

Object equivalence test.

Definition: <Object>

This message tests whether the receiver and the `comparand` are equivalent objects at the time the message is processed. Return *true* if the receiver is equivalent to `comparand`. Otherwise return *false*.

The meaning of "equivalent" cannot be precisely defined but the intent is that two objects are considered equivalent if they can be used interchangeably. Conforming protocols may choose to more precisely define the meaning of "equivalent".

The value of

```
receiver = comparand
```

is *true* if and only if the value of

```
comparand = receiver
```

would also be *true*. If the value of

```
receiver = comparand
```

is *true* then the receiver and `comparand` must have *equivalent hash values*. Or more formally:

```
receiver = comparand
receiver hash = comparand hash
```

The equivalence of objects need not be *temporally invariant*. Two independent invocations of #= with the same receiver and `operand` objects may not always yield the same results. Note that a collection that uses #= to discriminate objects may only reliably store objects whose hash values do not change while the objects are contained in the collection.

Refinement: <Character>

Two characters are considered equivalent if they have the same code point. In other words

```
character1 = character2
```

is *true* if and only if

```
character1 codePoint = character2 codePoint
```

is also *true*.

Parameters

`comparand` <Character> uncaptured

Return Value

<boolean> unspecified

Errors

none

Rationale

Note that object equality is not explicitly defined as being the same as object identity. That is probably the only reasonable implementation in Object but not specifying it allows this protocol to be used without refinement by other classes with more precise definitions of equivalence

5.3.4.2 Message: asLowercase

Synopsis

Return a character which is equivalent to the lowercase representation of the receiver.

Definition: <Character>

If the receiver is equal to the value of a character literal in the "receiver" row of the following table, the result object must be equal to the value of the corresponding character literal in the "result" row.

receiver	$A	$B	$C	$D	$E	$F	$G	$H	$I	$J	$K	$L	$M	$N	$O	$P	$Q	$R	$S	$T	$U	$V	$W	$X	$Y	$Z
result	$a	$b	$c	$d	$e	$f	$g	$h	$i	$j	$k	$l	$m	$n	$o	$p	$q	$r	$s	$t	$u	$v	$w	$x	$y	$z

An implemention may define other #asLowercase mappings. If the receiver does not correspond to a character in the "receiver" row of the table and does not have an implementation defined mapping the receiver is returned as the result.

Return Value

<Character> unspecified

Errors

none

5.3.4.3 Message: asString

Synopsis

Return a new string whose sole element is equivalent to the receiver.

Definition: <Character>

Return a new string of size one (1) whose sole element is equivalent to the receiver. The new string is created using the same constraints as defined by the `#new:` message defined in <String factory>. It is unspecified whether the resulting string captures a reference to the receiver.

Return Value

<String> new

Errors

none

5.3.4.4 Message: asUppercase

Synopsis

Return a character equivalent to the uppercase representation of the receiver.

Definition: <Character>

If the receiver is equal to the value of a character literal in the "receiver" row of the following table, the result object must be equal to the value of the corresponding character literal in the "result" row.

receiver	$a	$b	$c	$d	$e	$f	$g	$h	$i	$j	$k	$l	$m	$n	$o	$p	$q	$r	$s	$t	$u	$v	$w	$x	$y	$z
result	$A	$B	$C	$D	$E	$F	$G	$H	$I	$J	$K	$L	$M	$N	$O	$P	$Q	$R	$S	$T	$U	$V	$W	$X	$Y	$Z

An implemention may define other #asUppercase mappings. If the receiver does not correspond to a character in the "receiver" row of the table and does not have an implementation defined mapping the receiver is returned as the result.

Return Value

<Character> unspecified

Errors

none

5.3.4.5 Message: codePoint

Synopsis

Return the encoding value of the receiver.

Definition: <Character>

Return the encoding value of the receiver in the implementation defined execution character set.

The following invariant must hold:

```
(charFactory codePoint: x) codePoint = x
```

where `charFactory` is an object that implements <Character factory> and `x` is an <integer>.

Return Value

<integer>unspecified

Errors

none

5.3.4.6 Message: isAlphaNumeric

Synopsis

Test whether the receiver is a letter or digit.

Definition: <Character>

Return *true* if the receiver is either a letter or digit. Otherwise return *false*. In other words

```
character isAlphaNumeric
```

is *true* if and only if either

```
character isLetter
```

is *true* or

```
character isDigit
```

is *true*.

Return Value

<boolean> unspecified

Errors

none

5.3.4.7 Message: isDigit

Synopsis

Test whether the receiver is a digit.

Definition: <Character>

Return *true* if the receiver represents a digit. Otherwise return *false*. The receiver is a digit if it is equal to the value of one of the following character literals:

```
$0 $1 $2 $3 $4 $5 $6 $7 $8 $9
```

Return Value

<boolean> unspecified

Errors

none

5.3.4.8 Message: isLetter

Synopsis

Test whether the receiver is a letter.

Definition: <Character>

Return *true* if the receiver corresponds to an alphabetic character, ignoring case. Otherwise return *false*. The receiver is an alphabetic character if it is equal to the value of one of the following character literals:

```
$A $B $C $D $E $F $G $H $I $J $K $L $M
$N $O $P $Q $R $S $T $U $V $W $X $Y $Z
$a $b $c $d $e $f $g $h $i $j $k $l $m
$n $o $p $q $r $s $t $u $v $w $x $y $z
```

Implementations may define other characters to be alphabetic characters. Any such characters will return *true* when set this message.

Return Value

<boolean> unspecified

Errors

none

5.3.4.9 Message: isLowercase

Synopsis

Test whether the receiver is a lowercase letter.

Definition: <Character>

Return *true* if the receiver corresponds to a lowercase letter. Otherwise return *false*. The receiver is an lowercase letter if it is equal to the value of one of the following character literals:

```
$a $b $c $d $e $f $g $h $i $j $k $l $m
$n $o $p $q $r $s $t $u $v $w $x $y $z
```

Implementations may define other characters to be lowercase characters. Any such characters will return *true* when set this message.

Return Value

<boolean> unspecified

Errors

none

5.3.4.10 Message: isUppercase

Synopsis

Test whether the receiver is an uppercase letter.

Definition: <Character>

Return *true* if the receiver corresponds to a uppercase letter. Otherwise return *false*. The receiver is an uppercase letter if it is equal to the value of one of the following character literals:

```
$A $B $C $D $E $F $G $H $I $J $K $L $M
$N $O $P $Q $R $S $T $U $V $W $X $Y $Z
```

Implementations may define other characters to be lowercase characters. Any such characters will return *true* when set this message.

Return Value

<boolean> unspecified

Errors

none

5.3.5 Protocol: <Character factory>

Conforms To

<Object>

Description

This protocol describes the behavior for accessing character objects.

Standard Globals

`Character`	Conforms to the protocol <Object>. Its language element type is unspecified. This global is a factory for for creating or accessing objects that conform to <Character>.

Messages

codePoint:
cr
lf
space
tab

5.3.5.1 Message: codePoint: integer

Synopsis

Return a character whose encoding value is `integer`.

Definition: <Character factory>

Return a character whose encoding value in the implementation defined execution character set is `integer`.

The result is undefined if the encoding value is not a valid encoding value in the implementation defined character set.

Parameters

`integer`	<integer>	unspecified

Return Value

<Character> unspecified

Errors

none

5.3.5.2 Message: cr

Synopsis

Return a character representing a carriage-return.

Definition: <Character factory>

Return a character representing a carriage-return. The code point of the resulting character is implementation defined.

Return Value

<Character> unspecified

Errors

none

5.3.5.3 Message: lf

Synopsis

Return a character representing a line feed.

Definition: <Character factory>

Return a character representing a line feed. The code point of the resulting character is implementation defined.

Return Value

<Character> unspecified

Errors

none

5.3.5.4 Message: space

Synopsis

Return a character representing a space.

Definition: <Character factory>

Return a character representing a space. The code point of the resulting character is implementation defined.

Return Value

<Character> unspecified

Errors

none

5.3.5.5 Message: tab

Synopsis

Return a character representing a tab.

Definition: <Character factory>

Return a character representing a tab. The code point of the resulting character is implementation defined.

Return Value

<Character> unspecified

Errors

none

5.3.6 Protocol: <failedMessage>

Conforms To

<Object>

Description

This protocol describes the behavior of objects that represent a message that was sent to an object, but was not understood by that object.

Messages

arguments
selector

5.3.6.1 Message: arguments

Synopsis

Answer the arguments of the message that could not be sent.

Definition: <failedMessage>

Return a collection containing the arguments of the message that could not be sent. The elements of the collection are ordered, from the first element to the last element, in the same order as the arguments of the message, from left to right. If the message had no arguments, the collection will be empty.

Return Value

<sequenceReadabledCollection> unspecified

Errors

none

5.3.6.2 Message: selector

Synopsis

Answer the selector of the message that could not be sent.

Definition: <failedMessage>

Answer the selector of the message that could not be sent.

Return Value

<selector> unspecified

Errors

none

5.3.7 Protocol: <selector>

Conforms To

<Object>

Description

Defines the protocol supported by literal message selectors. No behavior is defined by this protocols but objects that conform to is can be used to perform dynamically generated message sends using <Object> #perform: and related messages.

Messages

none

5.3.8 Protocol: <classDescription>

Conforms To

<Object>

Description

This protocol describes the behavior of *class objects*. It provides messages for identifying and locating *class objects* within the class hierarchy.

Messages

allSubclasses
allSuperclasses
name
subclasses
superclass

Rationale

There are a wide variety of messages that various implementations provide for class objects. Most of them have been excluded from this definition because they are primarily oriented towards supporting a self-hosted development environment, and are not generally useful in non-reflective applications.

5.3.8.1 Message: allSubclasses

Synopsis

Return all subclasses of a class.

Definition: <classDescription>

If the receiver is a *class object*, return a collection containing all of the *class objects* whose *class definitions* inherit either directly or indirectly from the *class definition* of the receiver.

If the receiver is not a *class object*, the result is unspecified.

Each element of the result collection supports the protocol <classDescription>. The order of *class objects* within the collection is unspecified.

<collection> unspecified

Errors

none

5.3.8.2 Message: allSuperclasses

Synopsis

Return all superclasses of a class.

Definition: <classDescription>

If the receiver is a *class object*, return a collection containing all of the *class objects* defined by the *class definitions* from which the *class definition* of the receiver inherits, either directly or indirectly. If the *class definition* of the receiver has no superclasses, return an empty collection.

If the receiver is not a *class object*, the result is unspecified.

Each element of the result collection supports the protocol <classDescription>. The order of *class objects* within the collection is unspecified.

Return Value

<collection> unspecified

Errors

none

5.3.8.3 Message: name

Synopsis

Return the name of a class.

Definition: <classDescription>

Return a string containing the global name of the receiver. The global name of a *class object* is the global identifier that is bound to the *class object*.

Rationale

Some existing implementations may return a symbol as the result of this message. The specification of the return value should be whatever protocol is general enough to be either a string or a symbol.

Return Value

<readableString> unspecified

Errors

none

5.3.8.4 Message: subclasses

Synopsis

Return direct subclasses of a class.

Definition: <classDescription>

If the receiver is a *class object*, return a collection containing all of the *class objects* whose *class definitions* inherit directly from the *class definition* of the receiver. If there are no *class definitions* that inherit from the *class definition* of the receiver, return an empty collection.

If the receiver is not a *class object*, the result is unspecified.

Each element of the result collection supports the protocol <classDescription>. The order of *class objects* within the collection is unspecified.

Return Value

<collection> unspecified

Errors

none

5.3.8.5 Message: superclass

Synopsis

Return the immediate superclass of a class.

Definition: <classDescription>

If the receiver is a *class object*, return the *class objects* defined by the *class definitions* from which the *class definition* of the receiver directly inherits. If the *class definition* of the receiver has no superclasses, return *nil*.

If the receiver is not a *class object*, the result is unspecified.

Return Value

<classDescription> unspecified

<nil> unspecified

Errors

none

5.3.9 Protocol: <instantiator>

Conforms To

<Object>

Description

This protocol defines the behavior of objects that can be used to create other objects without requiring any additional information.

Messages

new

5.3.9.1 Message: new

Synopsis

Create a new object.

Definition: <instantiator>

Return a newly created object initialized to a standard initial state.

Return Value

<Object> new

Errors

none

5.3.10 Protocol: <Object class>

Conforms To

<classDescription>, <instantiator>

Description

This protocol describes the behavior the *class object* whose global identifier is 'Object', which is the traditional root of the class hierarchy.

This class must be implemented in such a way that it is not fragile. A class is said to be fragile if it is implemented in such a way that subclasses of that class can change the behavior of any standard-specified method without overriding the implementation of those methods. This can happen when a method is implemented to use an auxiliary method that is not specified in the standard, which the subclass then (possibly unintentionally) overrides. The inherited method will then invoke the subclass' implementation of the auxiliary method rather than the expected implementation in the superclass.

One way to ensure that the implementation of a class is not fragile is to ensure that any message sent to self is either part of the specified behavior for that class or has a selector that begins with an underscore. Alternatively, an implementation may use implementation-specific means to implement these methods in a way that makes them non-fragile.

Standard Globals

`Object` Conforms to the protocol <Object class>. It is a class object and the name of a class definition.

Messages

new

5.3.10.1 Message Refinement: new

Synopsis

Create a new object.

Definition: <instantiator>

Return a newly created object initialized to a standard initial state.

Refinement: <Object class>

Return a newly created instance of the receiver.

Return Value

<Object> new

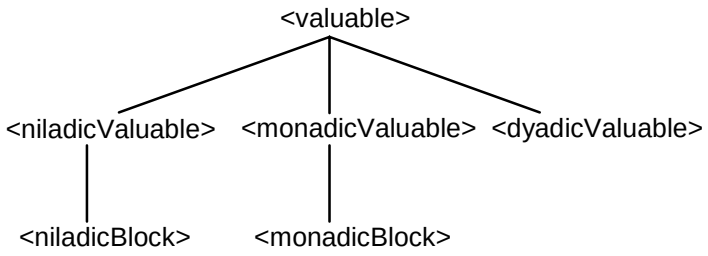
Errors

none

5.4 Valuable Protocols

This section includes protocols that describe objects that can be evaluated using variants of the `#value` message. The only concrete case of such objects specified by the standard are blocks. However, protocols that specify valuable protocols as parameters are defined to accept any class of object conforming to the specified protocol.

The graph below shows the conformance relationships between the protocols in this section.



5.4.1 Protocol: <valuable>

Conforms To

<Object>

Description

This protocol describes the behavior for objects that can be evaluated using variants of the `#value` message.

Rationale

chose to use selector `#ifCurtailed:` because of objections to `#ifTruncated:` and `#ifTerminated:`.as to suggest of process management operations.

Messages

argumentCount
valueWithArguments:

5.4.1.1 Message: argumentCount

Synopsis

Answers the number of arguments needed to evaluate the receiver.

Definition: <valuable>

The number of arguments needed to evaluate the receiver is returned.

Return Value

<integer>unspecified

Errors

none

5.4.1.2 Message: valueWithArguments: argumentArray

Synopsis

Answers the *value* of the receiver when applied to the arguments in `argumentArray`.

Definition: <valuable>

The receiver is evaluated as defined by the receiver.

Note that in the case that the receiver is a *block*, that the evaluation is defined by the language with the elements of `argumentArray` bound in sequence to the receiver's arguments.

The result is as defined by the receiver.

The results are undefined if the size of `argumentArray` does not equal the receiver's argument count.

Parameters

`argumentArray` <sequencedReadableCollection> uncaptured

Return Value

<ANY> unspecified

Errors

none

5.4.2 Protocol: <niladicValuable>

Conforms To

<valuable>

Description

This protocol describes the behavior for objects supporting the `#value` selector.

Messages

argumentCount
value
whileFalse
whileFalse:
whileTrue
whileTrue:

5.4.2.1 Message Refinement: argumentCount

Synopsis

Answers the number of arguments needed to evaluate the receiver.

Definition: <valuable>

The number of arguments needed to evaluate the receiver is returned.

Refinement: <niladicValuable>

Returns 0.

Return Value

<integer> unspecified

Errors

none

5.4.2.2 Message: value

Synopsis

Answers the *value* of the receiver.

Definition: <niladicValuable>

The receiver is evaluated as defined by the receiver.

The result is as defined by the receiver.

Return Value

<ANY> unspecified

Errors

none

5.4.2.3 Message: whileFalse

Synopsis

Evaluates the receiver until it evaluates to *true*.

Definition: <niladicValuable>

The receiver is evaluated as defined by the receiver.

Note that in the case that the receiver is a *block*, the evaluation is defined by the language.

If this evaluation results in *false* the process repeats.

If and when the evaluation of the receiver results in *true*, the method terminates.

The results are undefined if the receiver is not a *block* which evaluates to a Boolean value.

Return Value

UNSPECIFIED

Errors

none

5.4.2.4 Message: whileFalse: iterationBlock

Synopsis

Evaluates `iterationBlock` zero or more times until the receiver evaluates to *true*.

Definition: <niladicValuable>

The receiver is evaluated as defined by the receiver.

Note that in the case that the receiver is a *block*, that the evaluation is defined by the language.

If this evaluation results in *false*, the argument is evaluated and the process repeats.

If and when the evaluation of the receiver results in *true*, the method terminates.

The results are undefined if the receiver is not a *block* which evaluates to a Boolean value.

Parameters

`iterationBlock`	<niladicValuable>	uncaptured

Return Value

UNSPECIFIED

Errors

none

5.4.2.5 Message: whileTrue

Synopsis

Evaluates the receiver until it evaluates to *false*.

Definition: <niladicValuable>

The receiver is evaluated as defined by the receiver.

Note that in the case that the receiver is a *block*, that the evaluation is defined by the language.

If this evaluation results in *true* the process repeats.

If and when the evaluation of the receiver results in *false*, the method terminates.

The results are undefined if the receiver is not a *block* which evaluates to a Boolean value.

Return Value

UNSPECIFIED

Errors

none

5.4.2.6 Message: whileTrue: iterationBlock

Synopsis

Evaluates `iterationBlock` zero or more times until the receiver evaluates to *false*.

Definition: <niladicValuable>

The receiver is evaluated as defined by the receiver.

Note that in the case that the receiver is a *block*, that the evaluation is defined by the language.

If this evaluation results in *true*, the argument is evaluated and the process repeats.

If and when the evaluation of the receiver results in *false*, the method terminates.

The results are undefined if the receiver is not a *block* which evaluates to a Boolean value.

Parameters

`iterationBlock`	<niladicValuable>	uncaptured

Return Value

UNSPECIFIED

Errors

none

5.4.3 Protocol: <niladicBlock>

Conforms To

<niladicValuable>

Description

This protocol describes the behavior for blocks with no arguments.

Objects conforming to this protocol can be created only by the block constructor construct of the Smalltalk language.

Messages

ensure:
ifCurtailed:
on:do:

5.4.3.1 Message: ensure: terminationBlock

Synopsis

Evaluate a termination block after evaluating the receiver.

Definition: <niladicBlock>

Evaluate the receiver and return its result. Immediately after successful evaluation of the receiver but before returning its result, evaluate `terminationBlock`. If *abnormal termination* of the

receiver occurs, `terminationBlock` is evaluated. In either case, the value returned from the evaluation of `terminationBlock` is discarded.

Activation of an exception handler from within the receiver is not in and of itself an *abnormal termination*. However, if the *exception handler* for an exception that is not *resumable* results in termination of the receiver or if its *handler block* contains a return statement that results in *abnormal termination* of the receiver, then `terminationBlock` will be evaluated after evaluation of the *exception handler*.

If an *abnormal termination* results in the termination of multiple blocks which were evaluated using either `#ensure:` or `#ifCurtailed:` the respective `terminationBlocks` will be executed in the reverse of the order in which the corresponding receiver blocks were evaluated.

Parameters

`terminationBlock` <niladicBlock> uncaptured

Return Value

<ANY> unspecified

Errors

none

5.4.3.2 Message: ifCurtailed: terminationBlock

Synopsis

Evaluating the receiver with an abnormal termination action.

Definition: <niladicBlock>

Evaluate the receiver and return its result. If *abnormal termination* of the receiver occurs, `terminationBlock` is evaluated. The value returned from the evaluation of `terminationBlock` is discarded.

Activation of an exception handler from within the receiver is not in and of itself an *abnormal termination*. However, if the *exception handler* for an exception that is not *resumable* results in termination of the receiver or if its *handler block* contains a return statement that results in *abnormal termination* of the receiver, then `terminationBlock` will be evaluated after evaluation of the *exception handler*.

If an *abnormal termination* result in the termination of multiple blocks which were evaluated using either `#ensure:` or `#ifCurtailed:` the respective `terminationBlocks` will be executed in the reverse of the order in which the corresponding receiver blocks were evaluated.

Parameters

`terminationBlock` <niladicBlock> uncaptured

Return Value

<ANY> unspecified

Errors

none

5.4.3.3 Message: on: selector do: action

Synopsis

Evaluate the receiver in the scope of an exception handler.

Definition: <niladicBlock>

The receiver is evaluated such that if during its evaluation an exception corresponding to `selector` is signaled then `action` will be evaluated. The result of evaluating the receiver is returned.

Before evaluating the receiver the current state of the *exception environment* is captured as the *handler environment*. Then a new *exception handler* is created with `selector` as its *exception selector* and `action` as its *handler block*. The new handler is pushed onto the *exception environment*.

If evaluation of the receiver terminates normally then the *exception environment* is reset to the *handler environment* before returning to the sender of the `#on:do:` message.

If signaling of an exception results in evaluation of `action` the evaluation will occur in the context of the *handler environment*. The argument to the action will be an object that conforms to the protocol <signaledException>.

Parameters

`selector`	<exceptionSelector>	uncaptured
`action`	<monadicBlock>	uncaptured

Return Value

<ANY> unspecified

Errors

none

5.4.4 Protocol: <monadicValuable>

Conforms To

<valuable>

Description

This protocol describes the behavior for objects supporting the value: selector.

Messages

argumentCount
value:

5.4.4.1 Message Refinement: argumentCount

Synopsis

Answers the number of arguments needed to evaluate the receiver.

Definition: <valuable>

The number of arguments needed to evaluate the receiver is returned.

Refinement: <monadicValuable>

Returns 1.

Return Value

<integer> unspecified

Errors

none

5.4.4.2 Message: value: argument

Synopsis

Answers the *value* of the receiver when applied to the argument.

Definition: <monadicValuable>

The receiver is evaluated as defined by the receiver.

Note that in the case that the receiver is a *block*, that the evaluation is defined by the language with `argument` bound to the *block*'s only argument.

The result is as defined by the receiver.

Parameters

`argument` <ANY> unspecified

Return Value

<ANY> unspecified

Errors

none

5.4.5 Protocol: <monadicBlock>

Conforms To

<monadicValuable>

Description

This protocol describes the behavior for blocks with one argument.

Objects conforming to this protocol can be created only by the block constructor construct of the Smalltalk language.

Messages

none

5.4.6 Protocol: <dyadicValuable>

Conforms To

<valuable>

Description

This protocol describes the behavior for objects supporting the `#value:value:` selector.

Messages:

argumentCount

value:value:

5.4.6.1 Message Refinement: argumentCount

Synopsis

Answers the number of arguments needed to evaluate the receiver.

Definition: <valuable>

The number of arguments needed to evaluate the receiver is returned.

Refinement: <dyadicValuable>

Returns 2.

Return Value

<integer> unspecified

Errors

none

5.4.6.2 Message: value: argument1 value: argument2

Synopsis

Answers the value of the receiver when applied to the arguments.

Definition: <dyadic-valuable>

The receiver is evaluated as defined by the receiver.

Note that in the case that the receiver is a *block*, that the evaluation is defined by the language with `argument1` bound to the *block*'s first argument, and `argument2` bound to the *block*'s second argument.

The result is as defined by the receiver.

Parameters

`argument1` <ANY> unspecified

`argument2` <ANY> unspecified

Return Value

<ANY> unspecified

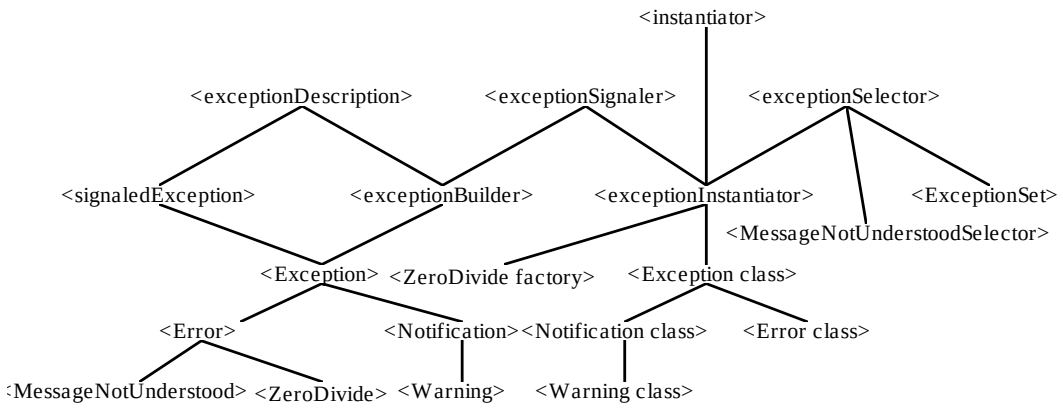
Errors

none

5.5 Exception Protocols

This section includes the protocols that define the behavior of the exception handling system.

The graph below shows the conformance relationships between the protocols in this section (except for the protocol <instantiator>, which is in the section containing fundamental protocols).



5.5.1 Protocol: <exceptionDescription>

Conforms To:

<Object>

Description

This protocol describe the messages that may be used to obtain information about an occurrence of an exception.

Messages

defaultAction
description
isResumable
messageText
tag

5.5.1.1 Message: defaultAction

Synopsis

The default action taken if the exception is signaled.

Definition: <exceptionDescription>

If the exception described by the receiver is signaled and the current *exception environment* does not contain a handler for the exception this method will be executed.

The exact behavior and result of this method is implementation defined.

Return Value

<Object> unspecified

Errors

none

5.5.1.2 Message: description

Synopsis

Return a textual description of the exception.

Definition: <exceptionDescription>

Return text that describes in a human readable form an occurrence of an exception. If an explicit message text was provided by the signaler of the exception, that text should be incorporated into the description.

Return Value

<readable> unspecified

Errors

none

5.5.1.3 Message: isResumable

Synopsis

Determine whether an exception is resumable.

Definition: <exceptionDescription>

This message is used to determine whether the receiver is a *resumable* exception. Answer *true* if the receiver is *resumable*. Answer *false* if the receiver is not *resumable*.

Return Value

<boolean> unspecified

Errors

none

5.5.1.4 Message: messageText

Synopsis

Return an exception's message text.

Definition: <exceptionDescription>

Return the signaler message text of the receiver. If the signaler has not provided any message text, return *nil*.

Return Value

<readableString> unspecified

<nil> unspecified

Errors

none

5.5.1.5 Message: tag

Synopsis

Return an exception's tag value.

Definition: <exceptionDescription>

Return the tag value provided by the signaler of the receiver. If the signaler has not provided a tag value, return the same value was would be returned as if #message Text was sent to the receiver of this message. If the signaler has provided neither a tag value nor a message text, return *nil*.

Exception tags are intended for use in situations where a particular occurrence of an exception needs to be identified and a textual description is not appropriate. For example, the message text might vary according to the locale and thus could not be used to identify the exception.

Return Value

<Object> unspecified

<nil> unspecified

Errors

none

5.5.2 Protocol: <exceptionSignaler>

Conforms To:

<Object>

Description

This protocol describes the behavior of signaling an exceptional condition, locating an *exception handler*, and executing an *exception action*.

Messages

signal

signal:

5.5.2.1 Message: signal

Synopsis

Signal the occurrence of an exceptional condition.

Definition: <exceptionSignaler>

Associated with the receiver is an <exceptionDescription> called the *signaled exception*. The current *exception environment* is searched for an *exception handler* whose *exception selector* matches the *signaled exception*. The search proceeds from the most recently created *exception handler* to the oldest *exception handler*.

A matching handler is defined to be one which would return *true* if the message `#handles:` was sent to its *exception selector* with the *signaled exception* as the argument.

If a matching handler is found, the *exception action* of the handler is evaluated in the *exception environment* that was current when the handler was created and the state of the current *exception environment* is preserved as the *signaling environment*.

The *exception action* is evaluated as if the message `#value:` were sent to it with a <signaledException> passed as its argument. The <signaledException> is derived from the *signaled exception* in an implementation dependent manner.

If the evaluation of the *exception action* returns normally (as if it had returned from the `#value: message`), the *handler environment* is restored and the value returned from the *exception action* is returned as the value of the `#on:do:` message that created the handler. Before returning, any active `#ensure:` or `#ifCurtailed:` termination blocks created during evaluation of the receiver of the `#on:do:` message are evaluated.

If a matching handler is not found when the *exception environment* is searched, the *default action* for the *signaled exception* is performed. This is accomplished as if the message `#defaultAction` were sent to the <signaledException> object derived from the *signaled exception*. The `#defaultAction` method is executed in the context of the *signaling environment*. If the *signaled exception* is *resumable* the value returned from the `#defaultAction` method is returned as the value of the `#signal` message. If the *signaled exception* is not *resumable* the action taken upon completion of the #defaultAction method is implementation defined.

Return Value

<Object> unspecified

Errors

none

5.5.2.2 Message: signal: signalerText

Synopsis

Signal the occurrence of an exceptional condition with a specified textual description.

Definition: <exceptionSignaler>

Associated with the receiver is an <exceptionDescription> called the *signaled exception*. The message text of the signaled exception is set to the value of `signalerText`, and then the exception is signaled in the same manner as if the message `#signal` had been sent to the receiver.

Note that this message does not return in some circumstances. The situations in which it does return and the returned value, if any, are the same as specified for the `#signal` message.

Parameters

`signalerText`	<readableString>	unspecified

Return Value

<Object> unspecified

Errors

none

5.5.3 Protocol: <exceptionBuilder>

Conforms To:

<exceptionDescription>, <exceptionSignaler>

Description

This protocol describes the messages that may be used to set the information about an occurrence of an exception. This information may be retrieved using <exceptionDescription> protocol. If an object conforming to this protocol is signaled as an exception, any information set in that object using this protocol's messages will also be available for retrival from the *signaled exception* that is passed to a *handler block*.

Messages

messageText:

5.5.3.1 Message: messageText: signalerText

Synopsis

Set an exception's message text.

Definition: <exceptionBuilder>

Set the signaler message text of the receiver. Subsequent sends of the message `#messgeText` to the receiver will return this value. Subseqent sends of of the message `#messgeText` to a *signaled exception* generated by sending the message `#signal` to the receiver of this message will also return this value.

Return the receiver as the result of the message.

Parameters

`signalerText` <readableString> captured

Return Value

<RECEIVER> unspecified

Errors

none

5.5.4 Protocol: <signaledException>

Conforms To:

<exceptionDescription>

Description

This protocol describes the messages that may be sent to the argument of a *handler block*. These message are used to explicitly control how execution will continue when it leaves the *handler block*.

Messages

isNested
outer
pass
resignalAs:
resume
resume:
retry
retryUsing:
return
return:

5.5.4.1 Message: isNested

Synopsis

Determine whether the current exception handler is within the scope of another handler for the same exception.

Definition: <signaledException>

Answer *true* if the *handler environment* for the current exception handler contains an *exception handler* that will *handle* the receiver. Answer *false* if it does not.

The *default action* for an exception is not considered to be an enclosing handler. Only the existence of a handler explicitly established using `#on:do:` will result in this method returning *true*.

Return Value

<boolean> unspecified

Errors

none

5.5.4.2 Message: outer

Synopsis

Evaluate the enclosing *exception action* for the receiver and return.

Definition: <signaledException>

If the *handler environment* for the current exception handler contains an *exception handler* that will *handle* the receiver, evaluate that handler's *exception action* with the receiver as the argument to its *handler block*. If there is no enclosing handler, send the message `#defaultAction` to the receiver. The #defaultAction method is evaluated using the current *exception environment*.

If the receiver is *resumable* and the evaluated *exception action* resumes then the result returned from `#outer` will be the *resumption value* of the evaluated *exception action*. If the receiver is not *resumable* or if the *exception action* does not resume then this message will not return.

For exceptions that are not *resumable*, `#outer` is equivalent to `#pass`.

Return Value

<Object> unspecified

Errors

It is erroneous to directly or indirectly send this message from within a <exceptionDescription>`#defaultAction` method to the receiver of the `#defaultAction` message.

5.5.4.3 Message: pass

Synopsis

Yield control to the enclosing *exception action* for the receiver.

Definition: <signaledException>

If the *handler environment* for the current exception handler contains an enclosing *exception handler* for the receiver, activate that handler's *exception action* in place of the current *exception action*. If there is no enclosing handler, execute the *default action* for the receiver as if no handler had been found when the exception was originally signaled. The default action is evaluated in the context of the signaling environment.

Control does not return to the currently active *exception handler*.

Return Value

none

Errors

It is erroneous to directly or indirectly send this message from within a `#defaultAction` method to the receiver of the `#defaultAction` method.

5.5.4.4 Message: resignalAs: replacementException

Synopsis

Signal an alternative exception in place of the receiver.

Definition: <signaledException>

The active *exception action* is aborted and the *exception environment* and the *evaluation context* are restored to the same states that were in effect when the receiver was originally signaled. Restoring the *evaluation context* may result in the execution of `#ensure:` or `#ifCurtailed:` termination blocks.

After the restoration, signal the `replacementException` and execute the *exception action* as determined by the restored *exception environment*.

This message causes the `replacementException` to be treated as if it had been originally signaled instead of the receiver.

If the `replacementException` is *resumable* and its *exception action* resumes, control will ultimately return from the message that signaled the original exception.

Control does not return from this message to the currently active *exception action*.

Parameters

`replacementException` <exceptionDescription> unspecified

Return Value

none

Errors

none

5.5.4.5 Message: resume

Synopsis

Return from the message that signaled the receiver.

Definition: <signaledException>

If the current *exception action* was activated as the result of sending the message #outer to the receiver, return a resumption value as the value of the #outer message.

If the receiver is a *resumable* exception a resumption value is returned as the value of the message that signaled the receiver. Before returning, the *exception environment* and the *evaluation context*

are restored to the same states that were in effect when the receiver was originally signaled. Restoring the *evaluation context* may result in the execution of `#ensure:` or `#ifCurtailed:` termination blocks.

This message does not return to its point of invocation.

The resumption value is unspecified.

Return Value

none

Errors

It is erroneous to directly or indirectly send this message from within a `#defaultAction` method to the receiver of the `#defaultAction` method.

It is erroneous to send the message if the receiver is not *resumable*.

5.5.4.6 Message: resume: resumptionValue

Synopsis

Return the argument as the value of the message that signaled the receiver.

Definition: <signaledException>

If the current *exception action* was activated as the result of sending the message #outer to the receiver, return `resumptionValue` as the value of the #outer message.

If the receiver is a *resumable* exception, the `resumptionValue` is returned as the value of the message that signaled the receiver. Before returning, the *exception environment* and the *evaluation context* are restored to the same states that were in effect when the receiver was originally signaled. Restoring the *evaluation context* may result in the execution of `#ensure:` or `#ifCurtailed:` termination blocks.

This message does not return to its point of invocation.

Parameters

`resumptionValue` `<Object>` uncaptured

Return Value

none

Errors

It is erroneous to directly or indirectly send this message from within a `#defaultAction` method to the receiver of the `#defaultAction` method.

It is erroneous to send the message if the receiver is not *resumable*.

5.5.4.7 Message: retry

Synopsis

Abort an exception handler and re-evaluate its *protected block*.

Definition: <signaledException>

The active *exception action* is aborted and the *exception environment* and the *evaluation context* are restored to the same states that were in effect when the `#on:do:` message that established the active handler was sent. Restoring the *evaluation context* may result in the execution of `#ensure:` or `#ifCurtailed:` termination blocks.

After the restoration, the `#on:do:` method is re-evaluated with its original receiver and arguments.

Control does not return from this message to the currently active *exception action*.

Return Value

none

Errors

It is erroneous to directly or indirectly send this message from within a `#defaultAction` method to the receiver of the `#defaultAction` method.

5.5.4.8 Message: retryUsing: alternativeBlock

Synopsis

Abort an exception handler and evaluate a new block in place of the handler's *protected block*.

Definition: <signaledException>

The active *exception action* is aborted and the *exception environment* and the *evaluation context* are restored to the same states that were in effect when the `#on:do:` message that established the active handler was sent. Restoring the *evaluation context* may result in the execution of `#ensure:` or `#ifCurtailed:` blocks.

After the restoration, the `#on:do:` method is re-evaluated with `alternativeBlock` substituted for its original receiver. The original arguments are used for the re-evaluation.

Control does not return from this message to the currently active *exception action*.

Parameters

`alternativeBlock` <niladicBlock> captured

Return Value

none

Errors

It is erroneous to directly or indirectly send this message from within a `#defaultAction` method to the receiver of the `#defaultAction` method.

5.5.4.9 Message: return

Synopsis

Return *nil* as the value of the block protected by the active exception handler.

Definition: <signaledException>

Nil is return as the value of the *protected block* of the active *exception handler*. Before returning, the *exception environment* and the *evaluation context* are restored to the same states that were in effect when the active handler was created using `#on:do:`. Restoring the *evaluation context* may result in the execution of `#ensure:` or `#ifCurtailed:` termination blocks.

This message does not return to its point of invocation.

Return Value

none

Errors

It is erroneous to directly or indirectly send this message from within a `#defaultAction` method to the receiver of the `#defaultAction` method.

5.5.4.10 Message: return: returnValue

Synopsis

Return the argument as the value of the block protected by the active exception handler.

Definition: <signaledException>

The `returnValue` is returned as the value of the *protected block* of the active *exception handler*. Before returning, the *exception environment* and the *evaluation context* are restored to the same states that were in effect when the active handler was created using `#on:do:`. Restoring the *evaluation context* may result in the execution of `#ensure:` or `#ifCurtailed:` termination blocks.

This message does not return to its point of invocation.

Parameters

`returnValue` <Object> uncaptured

Return Value

none

Errors

It is erroneous to directly or indirectly send this message from within a `#defaultAction` method to the receiver of the `#defaultAction` method.

5.5.5 Protocol: <exceptionSelector>

Conforms To:

<Object>

Description

This protocol describe the behavior of objects that are used to select an *exception handler*. In particular, objects that conform to this protocol may occur as the first argument to `#on:do:` message sent to blocks.

Messages

,
handles:

5.5.5.1 Message: , anotherException

Synopsis

Create an exception set.

Definition: <exceptionSelector>

Return an exception set that contains the receiver and the argument exception. This is commonly used to specify a set of exception selectors for an *exception handler*.

Parameters

`anotherException` <exceptionSelector> captured

Return Value

<exceptionSet> new

Errors

none

5.5.5.2 Message: handles: exception

Synopsis

Determine whether an *exception handler* will accept a *signaled exception*.

Definition: <exceptionSelector>

This message determines whether the *exception handler* associated with the receiver may be used to process the argument. Answer *true* if an associated handler should be used to process `exception`. Answer *false* if an associated handler may not be used to process the exception.

Parameters

`exception` <exceptionDescription> unspecified

Return Value

<boolean> unspecified

Errors

none

5.5.6 Protocol: <exceptionInstantiator>

Conforms To:

<exceptionSelector>, <exceptionSignaler>, <instantiator>

Description

This protocol describes the instantiation behavior of objects that can create exceptions.

Messages

new
signal

5.5.6.1 Message Refinement: new

Synopsis

Create a new object.

Definition: <instantiator>

Return a newly created object initialized to a standard initial state.

Refinement: <exceptionInstantiator>

The object returned is an <exceptionBuilder> that may be used to signal an exception of the same type that would be signaled if the message `#signal` is sent to the receiver.

Return Value

<exceptionBuilder> new

Errors

none

5.5.6.2 Message Refinement: signal

Synopsis

Signal the occurrence on an exceptional condition.

Definition: <exceptionSignaler>

Associated with the receiver is an <exceptionDescription> called the *signaled exception*. The current *exception environment* is searched for an *exception handler* whose *exception selector* matches the *signaled exception*. The search proceeds from the most recently created *exception handler* to the oldest *exception handler*.

A matching handler is defined to be one which would return *true* if the message `#handles:` was sent to its *exception selector* with the *signaled exception* as the argument.

If a matching handler is found, the *exception action* of the handler is evaluated in the *exception environment* that was current when the handler was created and the state of the current *exception environment* is preserved as the *signaling environment*.

The *exception action* is evaluated as if the message `#value:` were sent to it with a <signaledException> passed as its argument. The <signaledException> is derived from the *signaled exception* in an implementation dependent manner.

If the evaluation of the *exception action* returns normally (as if it had returned from the `#value: message`), the *handler environment* is restored and the value returned from the *exception action* is returned as the value of the `#on:do:` message that created the handler. Before returning, any active `#ensure:` or `#ifCurtailed:` termination blocks created during evaluation of the receiver of the `#on:do:` message are evaluated.

If a matching handler is not found when the *exception environment* is searched, the *default action* for the *signaled exception* is performed. This is accomplished as if the message `#defaultAction` were sent to the <signaledException> object derived from the *signaled exception*. The `#defaultAction` method is executed in the context of the *signaling environment*. If the *signaled exception* is *resumable* the value returned from the `#defaultAction` method is returned as the value of the `#signal` message. If the *signaled exception* is not *resumable* the action taken upon completion of the #defaultAction method is implementation defined.

Refinement: <exceptionInstantiator>

An exception of the type associated with the receiver is signaled. The <signaledException> is initialized to its default state.

Return Value

<Object> unspecified

Errors

none

5.5.7 Protocol: <Exception class>

Conforms To:

<classDescription> <exceptionInstantiator>

Description

This protocol describe the behavior of *class objects* that are used to create, signal, and select exceptions that exist within a specialization hierarchy.

The value of the standard global `Exception` is a class object that conforms to this protocol. The class `Exception` is explicitly specified to be subclassable. Conforming implementations must implement its behaviors in a non-*fragile* manner.

Standard Globals

`Exception`	A class name. Conforms to the protocol <Exception class>. Instances of this class conform to the protocol <Exception>.

Messages

handles:
new
signal

5.5.7.1 Message Refinement: handles: exception

Definition: <exceptionSelector>

This message determines whether the *exception handler* associated with the receiver may be used to process the argument. Answer *true* if an associated handler should be used to process `exception`. Answer *false* if an associated handler may not be used to process the exception.

Refinement: <Exception class>

Return *true* if the class of `exception` is the receiver or a general subclass of the receiver.

This definition implies that subclasses of an exception class are considered to be *subexception*s of the type of exception defined by their superclass. An *exception handler* that *handles* an exception class will also handle any exceptions that are instances of the exception class's subclasses.

Return Value

<boolean> unspecified

Errors

none

5.5.7.2 Message Refinement: new

Definition: <instantiator>

Return a newly created object initialized to a standard initial state.

Refinement: <exceptionInstantiator>

The object returned is an <exceptionBuilder> that may be used to signal an exception of the same type that would be signaled if the message `#signal` is sent to the receiver.

Refinement: <Exception class>

The object returned conforms to <Exception>

Return Value

<Exception> new

Errors

none

5.5.7.3 Message Refinement: signal

Definition: <exceptionSignaler>

Associated with the receiver is an <exceptionDescription> called the *signaled exception*. The current *exception environment* is searched for an *exception handler* whose *exception selector* matches the *signaled exception*. The search proceeds from the most recently created *exception handler* to the oldest *exception handler*.

A matching handler is defined to be one which would return *true* if the message `#handles:` was sent to its *exception selector* with the *signaled exception* as the argument.

If a matching handler is found, the *exception action* of the handler is evaluated in the *exception environment* that was current when the handler was created and the state of the current *exception environment* is preserved as the *signaling environment*.

The *exception action* is evaluated as if the message `#value:` were sent to it with a <signaledException> passed as its argument. The <signaledException> is derived from the *signaled exception* in an implementation dependent manner.

If the evaluation of the *exception action* returns normally (as if it had returned from the `#value: message`), the *handler environment* is restored and the value returned from the *exception action* is returned as the value of the `#on:do:` message that created the handler. Before returning, any active `#ensure:` or `#ifCurtailed:` termination blocks created during evaluation of the receiver of the `#on:do:` message are evaluated.

If a matching handler is not found when the *exception environment* is searched, the *default action* for the *signaled exception* is performed. This is accomplished as if the message `#defaultAction` were sent to the <signaledException> object derived from the *signaled exception*. The `#defaultAction` method is executed in the context of the *signaling environment*. If the *signaled exception* is *resumable* the value returned from the `#defaultAction` method is returned as the value of the `#signal` message. If the *signaled exception* is not *resumable* the action taken upon completion of the #defaultAction method is implementation defined.

Refinement: <Exception class>

The exception signaled conforms to <Exception> with all of its <exceptionDescription> attributes set to their default values.

Return Value

<Object> unspecified

Errors

none

5.5.8 Protocol: <Exception>

Conforms To:

<exceptionBuilder>, <signaledException>

Description

This protocol describes the behavior of instances of class `Exception`. Typically, actual exceptions used by an application will be either direct or indirect subclasses of this class. Exception combines the behavior of <exceptionBuilder> and <signaledException>. Instances are used to both supplied inform before an exception is signaled and to pass the information to an exception handler.

As `Exception` is explicitly specified to be subclassable, conforming implementations must implement its behavior in a non-*fragile* manner.

Rationale

Exception is an abstract class. It is the only true abstract class specified by the standard. It is included so as to provide a mechanism for the protable definition of new exception. Exceptions defined as subclasses of Exception will be portable to any conforming implementation..

Messages

none

5.5.9 Protocol: <Notification class>

Conforms To:

<Exception class>

Description

This protocol describe the behavior of the global `Notification`. The value of the standard global `Notification` is a class object that conforms to this protocol. The class `Notification` is explicitly specified to be subclassable in a standard conforming program. Conforming implementations must implement its behaviors in a non-*fragile* manner.

The *signaled exceptions* generated by this type of object conform to the protocol <Notification>.

Standard Globals

`Notification`	A class name. Conforms to the protocol <Notification class>. `Notification` must inherit (possibly indirectly) from the class `Exception`. Instances of this class conform to the protocol <Notification>.

Messages

new

5.5.9.1 Message Refinement: new

Definition: <instantiator>

Return a newly created object initialized to a standard initial state.

Refinement: <exceptionInstantiator>

The object returned is an <exceptionBuilder> that may be used to signal an exception of the same type that would be signaled if the message `#signal` is sent to the receiver.

Refinement: <Exception class>

The object returned conforms to <Exception>

Refinement: <Notification class>

The object returned conforms to <Notification>.

Return Value

<Notification> new

Errors

none

5.5.10 Protocol: <Notification>

Conforms To:

<Exception>

Description

This protocol describes the behavior of instances of the class `Notification`. These are used to represent exceptional conditions that may occur but which are not considered errors. Actual notification exceptions used by an application may be subclasses of this class.

As `Notification` is explicitly specified to be subclassable, conforming implementations must implement its behavior in a non-*fragile* manner.

Messages

defaultAction
isResumable

5.5.10.1 Message Refinement: defaultAction

Definition: <exceptionDescription>

If the exception described by the receiver is signaled and the current *exception environment* does not contain a handler for the exception this method will be executed.

The exact behavior and result of this method is implementation defined.

Refinement: <Notification>

No action is taken. The value *nil* is returned as the value of the message that signaled the exception.

Return Value

<nil> unspecified

Errors

none

5.5.10.2 Message Refinement: isResumable

Definition: <exceptionDescription>

This message is used to determine whether the receiver is a *resumable* exception. Answer *true* if the receiver is *resumable*. Answer *false* if the receiver is not *resumable*.

Refinement: <Notification>

Answer *true*. Notification exceptions by default are specified to be *resumable*.

Return Value

<boolean> unspecified

Errors

none

5.5.11 Protocol: <Warning class>

Conforms To:

<Notification class>

Description

This protocol describe the behavior of the global `Warning`. The value of the standard global `Warning` is a class object that conforms to this protocol. The class `Warning` is explicitly specified to be subclassable in a standard conforming program. Conforming implementations must implement its behaviors in a non-*fragile* manner.

The *signaled exceptions* generated by this type of object conform to the protocol <Warning>.

Standard Globals

`Warning`	A class name. Conforms to the protocol <Warning class>. `Warning` must inherit (possibly indirectly) from the class `Notification`. Instances of this class conform to the protocol <Warning>.

Messages

new

5.5.11.1 Message Refinement: new

Definition: <instantiator>

Return a newly created object initialized to a standard initial state.

Refinement: <exceptionInstantiator>

The object returned is an <exceptionBuilder> that may be used to signal an exception of the same type that would be signaled if the message `#signal` is sent to the receiver.

Refinement: <Exception class>

The object returned conforms to <Exception>

Refinement: <Warning class>

The object returned conforms to <Warning>

Return Value

<Warning> new

Errors

none

5.5.12 Protocol: <Warning>

Conforms To:

<Notification>

Description

This protocol describes the behavior of instances of class Warning. These are used to represent exceptional conditions that might occur that are not considered errors but which should be reported to the user. Typically, the actual warning exceptions used by an application will be subclasses of this class.

As `Warning` is explicitly specified to be subclassable, conforming implementations must implement its behavior in a non-*fragile* manner.

Messages

defaultAction

5.5.12.1 Message Refinement: defaultAction

Synopsis

The default action taken if the exception is signaled.

Definition: <exceptionDescription>

If the exception described by the receiver is signaled and the current *exception environment* does not contain a handler for the exception this method will be executed.

The exact behavior and result of this method is implementation defined.

Refinement: <Notification>

No action is taken. The value *nil* is returned as the value of the message that signaled the exception.

Refinement: <Warning>

The user should be notified of the occurrence of an exceptional occurrence and given an option of continuing or aborting the computation. The description of the occurrence should include any text specified as the argument of the `#signal:` message.

Return Value

UNSPECIFIED

Errors

none

5.5.13 Protocol: <Error class>

Conforms To:

<Exception class>

Description

This protocol describe the behavior of the global `Error`. The value of the standard global `Error` is a class object that conforms to this protocol. The class `Error` is explicitly specified to be subclassable in a standard conforming program. Conforming implementations must implement its behaviors in a non-*fragile* manner.

The *signaled exceptions* generated by this type of object conform to the protocol <Error>.

Standard Globals

`Error`	A class name. Conforms to the protocol <Error class>. `Error` must inherit (possibly indirectly) from the class `Exception`. Instances of this class conform to the protocol <Error>.

Messages

new

5.5.13.1 Message Refinement: new

Definition: <instantiator>

Return a newly created object initialized to a standard initial state.

Refinement: <exceptionInstantiator>

The object returned is an <exceptionBuilder> that may be used to signal an exception of the same type that would be signaled if the message `#signal` is sent to the receiver.

Refinement: <Exception class>

The object returned conforms to <Exception>

Refinement: <Error class>

The object returned conforms to <Error>

Return Value

<Error> new

Errors

none

5.5.14 Protocol: <Error>

Conforms To:

<Exception>

Description

This protocol describes the behavior of instances of class Error. These are used to represent error conditions that prevent the normal continuation of processing. Actual error exceptions used by an application may be subclasses of this class.

As `Error` is explicitly specified to be subclassable, conforming implementations must implement its behavior in a non-*fragile* manner.

Messages

defaultAction
isResumable

5.5.14.1 Message Refinement: defaultAction

Definition: <exceptionDescription>

If the exception described by the receiver is signaled and the current *exception environment* does not contain a handler for the exception this method will be executed.

The exact behavior and result of this method is implementation defined.

Refinement: <Error>

The current computation is terminated. The cause of the error should be logged or reported to the user. If the program is operating in an interactive debugging environment the computation should be suspended and the debugger activated.

Return Value

UNSPECIFIED

Errors

none

5.5.14.2 Message Refinement: isResumable

Synopsis

Determine whether an exception is resumable.

Definition: <exceptionDescription>

This message is used to determine whether the receiver is a *resumable* exception. Answer *true* if the receiver is *resumable*. Answer *false* if the receiver is not *resumable*.

Refinement: <Error>

Answer *false*. Error exceptions by default are assumed to not be *resumable*. Subclasses may override this definition for situations where it is appropriate for an error to be resumable.

Return Value

<boolean> unspecified

Errors

none

5.5.15 Protocol: <ZeroDivide factory>

Conforms To:

<exceptionInstantiator>

Description

This protocol describe the behavior of the global `ZeroDivide`. It is used to as an *exception selector* to catch zero divide exceptions and can also be used to signal that a division by zero error has occured. Zero divide exceptions are resumable so any message in this protocol that signal such an exception may ultimately return to their sender. The *signaled exceptions* generated by this type of object conform to the protocol <ZeroDivide>

Standard Globals

`ZeroDivide`	Unspecified language element type. Conforms to the protocol <ZeroDivide class>.

Messages

dividend:
signal

5.5.15.1 Message: dividend: argument

Synopsis

Signal the occurance of a division by zero.

Refinement: <ZeroDivide factory>

Signal the occurance of a division by zero exception. Capture the number that was being divided such that it is available from the *signaled exception*.

If the message `#dividend` is subsequently sent to the <ZeroDivide> object that is the *signaled exception* the value of `argument` is returned.

Parameters

`argument`	<number> captured

Return Value

<Object>	state

Errors

none

5.5.15.2 Message Refinement: signal

Definition: <exceptionSignaler>

Associated with the receiver is an <exceptionDescription> called the *signaled exception*. The current *exception environment* is searched for an *exception handler* whose *exception selector* matches the *signaled exception*. The search proceeds from the most recently created *exception handler* to the oldest *exception handler*.

A matching handler is defined to be one which would return *true* if the message `#handles:` was sent to its *exception selector* with the *signaled exception* as the argument.

If a matching handler is found, the *exception action* of the handler is evaluated in the *exception environment* that was current when the handler was created and the state of the current *exception environment* is preserved as the *signaling environment*.

The *exception action* is evaluated as if the message `#value:` were sent to it with a <signaledException> passed as its argument. The <signaledException> is derived from the *signaled exception* in an implementation dependent manner.

If the evaluation of the *exception action* returns normally (as if it had returned from the `#value: message)`, the *handler environment* is restored and the value returned from the *exception action* is returned as the value of the `#on:do:` message that created the handler. Before returning, any active `#ensure:` or `#ifCurtailed:` termination blocks created during evaluation of the receiver of the `#on:do:` message are evaluated.

If a matching handler is not found when the *exception environment* is searched, the *default action* for the *signaled exception* is performed. This is accomplished as if the message `#defaultAction` were sent to the <signaledException> object derived from the *signaled exception*. The `#defaultAction` method is executed in the context of the *signaling environment*. If the *signaled exception* is *resumable* the value returned from the `#defaultAction` method is returned as the value of the `#signal` message. If the *signaled exception* is not *resumable* the action taken upon completion of the #defaultAction method is implementation defined.

Refinement: <exceptionInstantiator>

An exception of the type associated with the receiver is signaled. The <signaledException> is initialized to its default state.

Refinement: <ZeroDivide factory>

The *signaled exception* conforms to <ZeroDivide> and all of its <exceptionDescription> attributes set to their default values.

Return Value

<Object> unspecified

Errors

none

5.5.16 Protocol: <ZeroDivide>

Conforms To:

<Error>

Description

This protocol describes the behavior of exceptions that are signalled when an attempt is made to divide some number (the dividend) by zero.

Messages

dividend
isResumable

5.5.16.1 Message: dividend

Synopsis

Answer the number that was being divided by zero.

Definition: <ZeroDivide>

Answer the number that was being divided by zero.

Return Value

<number> state

Errors

none

5.5.16.2 Message Refinement: isResumable

Synopsis

Determine whether an exception is resumable.

Definition: <exceptionDescription>

This message is used to determine whether the receiver is a *resumable* exception. Answer *true* if the receiver is *resumable*. Answer *false* if the receiver is not *resumable*.

Refinement: <Error>

Answer *false*. Error exceptions by default are assumed to not be *resumable*. Subclasses may override this definition for situations where it is appropriate for an error to be resumable.

Refinement: <ZeroDivide>

Answer *true*.

Return Value

<boolean> unspecified

Errors

none

5.5.17 Protocol: <MessageNotUnderstoodSelector>

Conforms To:

<ExceptionSelector>

Description

This protocol describe the behavior of the value of the global named `MessageNotUnderstood`. This object is used to as an *exception selector* to catch failed message sends. Message not understood exceptions are resumable so any message in this protocol that signal such an exception may ultimately return to their sender.

This object is not specifed as an <exceptionSignaler> or an <exceptionInstantiator>. It as assumed that message not understood exceptions are signaled by the implemention dependent implementaton of the message <Object> #doesNotUnderstand:.

Standard Globals

`MessageNotUnderstood`
Unspecified language element type. Conforms to the protocol <MessageNotUnderstoodSelector>. Used as an *exception selector* .

Messages

handles:

5.5.17.1 Message Refinement: handles: exception

Synopsis

Determine whether an *exception handler* will accept a *signaled exception*.

Definition: <exceptionSelector>

This message determines whether the *exception handler* associated with the receiver may be used to process the argument. Answer *true* if an associated handler should be used to process `exception`. Answer *false* if an associated handler may not be used to process the exception.

Refinement: <MessageNotUnderstoodSelector>

Return true if `exception` is an exception that is the result of a failed message send.

Parameters

`exception` <exceptionDescription> unspecified

Return Value

<boolean> unspecified

Errors

none

5.5.18 Protocol: <MessageNotUnderstood>

Conforms To:

<Error>

Description

This protocol describes the behavior of exceptions that are signalled if the receiver of a message does not have a method with a matching selector.

Messages

message
isResumable
message
receiver

5.5.18.1 Message: message

Synopsis

Answer the selector and arguments of the message that failed.

Definition: <MessageNotUnderstood>

Answer the selector and arguments of the message that failed.

Return Value

<failedMessage> state

Errors

none

5.5.18.2 Message Refinement: isResumable

Synopsis

Determine whether an exception is resumable.

Definition: <exceptionDescription>

This message is used to determine whether the receiver is a *resumable* exception. Answer *true* if the receiver is *resumable*. Answer *false* if the receiver is not *resumable*.

Refinement: <Error>

Answer *false*. Error exceptions by default are assumed to not be *resumable*. Subclasses may override this definition for situations where it is appropriate for an error to be resumable.

Refinement: <MessageNotUnderstood>

Answer *true*.

Return Value

<boolean> unspecified

Errors

none

5.5.18.3 Message: message

Synopsis

Answer the selector and arguments of the message that failed.

Definition: <MessageNotUnderstood>

Answer the selector and arguments of the message that failed.

Return Value

<failedMessage> state

Errors

none

5.5.18.4 Message: receiver

Synopsis

Answer the receiver the message that failed.

Definition: <MessageNotUnderstood>

Answer the object that was the receiver of the message that failed.

Return Value

<Object> state

Errors

none

5.5.19 Protocol: <exceptionSet>

Conforms To:

<exceptionSelector>

Description

This protocol describes the behavior of objects that may be used to group a set of <exceptionSelector> objects into a single <exceptionSelector>. This is useful for establishing a single *exception handler* that may deal with several different types of exceptions.

Messages

,

5.5.19.1 Message Refinement: , anotherException

Definition: <exceptionSelector>

Return an exception set that contains the receiver and the argument exception. This is commonly used to specify a set of exception selectors for an *exception handler*.

Refinement: <exceptionSet>

In addition to `anotherException` the exception set that is returned contains all of the*exception selectors* contained in the receiver.

The returned object may or may not be the same object as the receiver.

Parameters

`anotherException` <exceptionSelector> captured

Return Value

<exceptionSet> unspecified

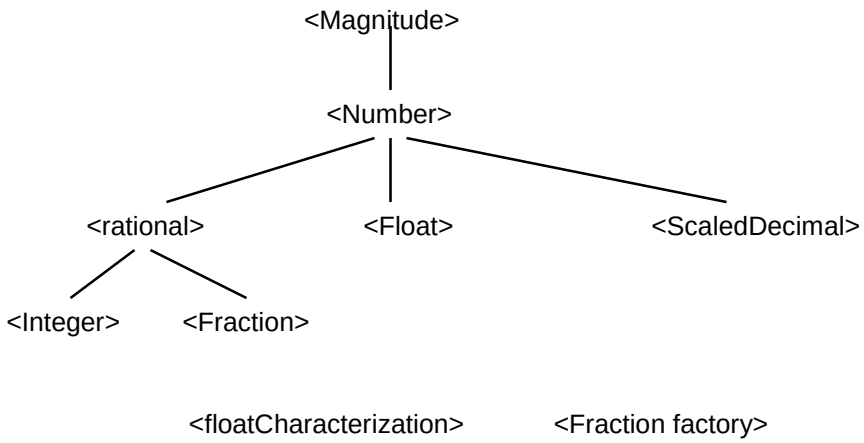
Errors

none

5.6 Numeric Protocols

This section includes protocols that define the behavior of the standard numeric classes.

The graphs below shows the conformance relationships between the protocols defined in this section, including <magnitude>. The protocols <factory> and <Object> are not part of this section.



5.6.1 Protocol: <magnitude>

Conforms To

<Object>

Description

Provides protocol for comparing objects which are linearly ordered with respect to some comparison operation.

Messages

<
<=
>
>=
between:and:
max:
min:

5.6.1.1 Message: < operand

Synopsis

Answer *true* if the receiver is less than operand. Answer *false* otherwise.

Definition: <magnitude>

Answer *true* if the receiver is less than operand with respect to the ordering defined for them. Answer *false* otherwise.

It is erroneous if the receiver and operand are not *comparable*.

The semantics of the natural ordering must be defined by refinement, which may also restrict the type of operand.

Parameters

operand <magnitude> uncaptured

Return Values

<boolean> unspecified

Errors

Receiver and operand are not *comparable*

5.6.1.2 Message: <= operand

Synopsis

Answer *true* if the receiver is less than or equal to operand. Answer *false* otherwise.

Definition: <magnitude>

Answer *true* if the receiver would answer *true* to either the #< or #= message with operand as the parameter. Answer *false* otherwise.

It is erroneous if the receiver and operand are not *comparable*.

Parameters

operand <magnitude> uncaptured

Return Values

<boolean> unspecified

Errors

Receiver and operand are not *comparable*

5.6.1.3 Message: > operand

Synopsis

Answer *true* if the receiver is greater than operand. Answer *false* otherwise.

Definition: <magnitude>

Answer *true* if the receiver is greater than operand with respect to the natural ordering. Answer *false* otherwise.

It is erroneous if the receiver and operand are not *comparable*.

The semantics of the natural ordering must be defined by refinement, which may also restrict the type of operand.

Parameters

operand <magnitude> uncaptured

Return Values

<boolean> unspecified

Errors

Receiver and operand are not *comparable*

5.6.1.4 Message: >= operand

Synopsis

Answer *true* if the receiver is greater than or equal to operand. Answer *false* otherwise.

Definition: <magnitude>

Answer *true* if the receiver answers *true* to either the #> or #= message with operand as the parameter. Answer *false* otherwise.

It is erroneous if the receiver and operand are not *comparable*.

Parameters

operand <magnitude> uncaptured

Return Values

<boolean> unspecified

Errors

Receiver and operand are not *comparable*

5.6.1.5 Message: between: min and: max

Synopsis

Answer *true* if the receiver is less than or equal to max, and greater than or equal to min. Answer *false* otherwise.

Definition: <magnitude>

Answer *true* if the receiver answers *true* to the #<= message with max as the parameter, and also answers *true* to the #>= message with min as the parameter. Answer *false* otherwise.

It is erroneous if the receiver and min or max are not *comparable*.

Parameters

min <magnitude> uncaptured

max <magnitude> uncaptured

Return Values

<boolean> unspecified

Errors

Receiver and operands are not *comparable*.

5.6.1.6 Message: max: operand

Synopsis

Answer the receiver if it is greater than operand. Answer operand otherwise.

Definition: <magnitude>

Answer the receiver if the receiver answers *true* to the #> message with operand as the parameter. Answer operand otherwise.

It is erroneous if the receiver and operand are not *comparable*.

Parameters

operand <magnitude> uncaptured

Return Values

<magnitude> unspecified

Errors

Receiver and operand are not *comparable*

5.6.1.7 Message: min: operand

Synopsis

Answer the receiver if it is less than operand. Answer operand otherwise.

Definition: <magnitude>

Answer the receiver if the receiver answers *true* to the #< message with operand as the parameter. Answer operand otherwise.

It is erroneous if the receiver and operand are not *comparable*.

Parameters

operand <magnitude> uncaptured

Return Values

<magnitude> unspecified

Errors

Receiver and operand are not *comparable*

5.6.2 Protocol: <number>

Conforms To

<magnitude>

Description

Provides protocol for objects that represent numeric quantities and support operations performing arithmetic, arithmetic progressions, and conversion on numerical quantities.

The descriptions of messages in this protocol reference specific arithmetic and numerical operations in the ISO/IEC 10967 standard, providing definition-by-reference for these operations.

Smalltalk provides for mixed-mode arithmetic with the receiver and argument having different *numeric representations*. Unless otherwise specified by an individual operation the receiver and argument are first converted to the same *numeric representation* according to the following table.

Default Conversion Table:

operand receiver	<integer>	<scaledDecimal>	<Fraction>	<Float>$_e$	<Float>$_d$	<Float>$_q$
<integer>	<integer>	<scaledDecimal>	<Fraction>	<Float>$_e$	<Float>$_d$	<Float>$_q$
<scaledDecimal>	<scaledDecimal>	<scaledDecimal>	<Fraction>	<Float>$_e$	<Float>$_d$	<Float>$_q$
<Fraction>	<Fraction>	<Fraction>	<Fraction>	<Float>$_e$	<Float>$_d$	<Float>$_q$
<Float>$_e$	<Float>$_e$	<Float>$_e$	<Float>$_e$	<Float>$_e$	<Float>$_d$	<Float>$_q$
<Float>$_d$	<Float>$_d$	<Float>$_d$	<Float>$_d$	<Float>$_d$	<Float>$_d$	<Float>$_q$
<Float>$_q$	<Float>$_q$	<Float>$_q$	<Float>$_q$	<Float>$_q$	<Float>$_q$	<Float>$_q$

If multiple representations of <Float> are available, the representations are ordered from smallest to largest precision. This table contains multiple entries for <Float>, designated by a subscript, one for each designation of floating point literal representation. Values that are converted to <Float> are converted to the smallest precision of Float that can represent the number of digits in the original value.

An <integer> converted to a <scaledDecimal> will have the scale of the other operand with the fractional digits set to zero. A <scaledDecimal> converted to a <Fraction> will be a fraction having the same numeric value but having an integer numerator and a denominator which is ten raised to the power of the <scaledDecimal>'s scale factor.

The result type of most numeric opeations is based upon the operaand type. The Default Result Type for all operand types except <Fraction> is the type to which the operands have been converted according to the Default ConversionTable. If the converted operand type is <Fraction> the Default Result Type is <rational>. In all cases where the type of the return value differs from the default result type it is noted in the operation's description.

Operations can produce results that are outside the set of representable numbers, or mathematically undefined. It is implementation defined whether errors are raised when results are not representable or if unrepresentable results are wrapped in implementation-defined continuation values or their equivalent. The effect of underflow and overflow is therefore implementation defined.

<number> conforms to <magnitude>. All object that implement the <number> protocol or any protocol that conforms to <number> are *comparable*.

Messages

*
+
- operand
//
<
=
>
\\
abs
asFloat
asFloatD
asFloatE
asFloatQ
asFraction
asInteger
asScaledDecimal:
ceiling
floor
fractionPart
integerPart
negated
negative

positive
printString
quo:
raisedTo:
raisedToInteger:
reciprocal
rem:
rounded
roundTo:
sign
sqrt
squared
strictlyPositive
to:
to:by:
to:by:do:
to:do:
truncated
truncateTo:

5.6.2.1 Message: * operand

Synopsis

Answer the result of multiplying the receiver by `operand`.

Definition: <number>

Answer a number whose value is the result of multiplying the receiver and `operand`, as specified by the ISO/IEC 10967 multiplication operation *mul*. To perform the operation both the receiver and `operand` must be objects with identical *numeric representations*. If they have different representations a conversion to their common *numeric representation* is performed, as specified by the Default Conversion Table, before applying the operation. If the common representation is <integer>, then the result value is defined by the ISO/IEC 10967 operation mul_I. If the common representation is <Float>, then the result value is defined by the ISO/IEC 10967 mul_F. Otherwise, the result is consistent with the mathematical definition of the ISO/IEC 10967 operation *mul*.

The protocol and representation of the return value is defined to be the Default Result Type. If the return value conforms to <scaledDecimal> then the scale of the result is at least the scale of the receiver after conversion if necessary.

If the result value is outside of the *range* of the common *numeric representation,* the effect of underflow or overflow is implementation defined..

Parameters

`operand` <number> unspecified

Return Values

The Default Result Type

Errors

none

5.6.2.2 Message: + operand

Synopsis

Answer the result of adding `operand` to the receiver.

Definition: <number>

Answer a number whose value is the result of adding the receiver and `operand`, as specified by the ISO/IEC 10967 addition operation *add*. To perform the operation both the receiver and `operand` must be objects with identical *numeric representations*. If they have different

representations a conversion to a common *numeric representation* is performed, as specified by the Default Conversion Table, before applying the operation. If the resulting protocol is <integer>, then the result value is defined by the ISO/IEC 10967 operation add_I. If the resulting protocol is <Float>, then the result value is defined by the ISO/IEC 10967 add_F. Otherwise, the result is consistent with the mathematical definition of the ISO/IEC 10967 operation *add*.

The protocol and representation of the return value is defined by the Default Result Type. If the return value conforms to <scaledDecimal> then the scale of the result is at least the scale of the receiver after conversion if necessary. If the result value is outside of the *range* of the common *numeric representation,* the effect of underflow or overflow is implementation defined.

Parameters

`operand` <number> unspecified

Return Values

The Default Result Type

Errors

none

5.6.2.3 Message: - operand

Synopsis

Answer the result of subtracting `operand` from the receiver.

Definition: <number>

Answer a number whose value is the result of subtracting the receiver and `operand`, as specified by the ISO/IEC 10967 subtraction operation *sub*. To perform the operation both the receiver and `operand` must be objects with identical *numeric representations*. If they have different representations a conversion to a common *numeric representation* is performed, as specified by the Default Conversion Table, before applying the operation. If the resulting protocol is <integer>, then the result value is defined by the ISO/IEC 10967 operation sub_I. If the resulting protocol is <Float>, then the result value is defined by the ISO/IEC 10967 sub_F. Otherwise, the result is consistent with the mathematical definition of the ISO/IEC 10967 operation *sub*.

The protocol and representation of the return value is defined by the Default Result Type. If the return value conforms to <scaledDecimal> then the scale of the result is at least the scale of the receiver after conversion if necessary. If the result value is outside of the *range* of the common *numeric representation,* the effect of underflow or overflow is implementation defined.

Parameters

`operand` <number> unspecified

Return Values

The Default Result Type

Errors

none

5.6.2.4 Message: / operand

Synopsis

Answer the result of dividing the receiver by `operand`.

Definition: <number>

Answer a number whose value is the result of dividing the receiver by `operand`, as specified by the ISO/IEC 10967 flooring division operation *div* To perform the operation both the receiver and `operand` must be objects with identical *numeric representations*. If they have different representations a conversion to a common *numeric representation* is performed, as specified by the Default Conversion Table, before applying the operation. If the resulting protocol is <integer>,

then the result value is a <rational> with the receiver as the numerator and the `operand` as the denominator. If the resulting protocol is <Float>, then the result value is defined by the ISO/IEC 10967 div_F. Otherwise, the result is consistent with the mathematical definition of the ISO/IEC 10967 operation *div*.

If both operands conform to <integer> the result value will conform to <rational>. Otherwise the protocol and representation of the return value are defined by the Default Result Type. If the return value conforms to <scaledDecimal> then the scale of the result is at least the scale of the receiver after conversion if necessary.

If the result value is outside of the *range* of the common *numeric representation,* the effect of underflow or overflow is implementation defined. If either the receiver or operand are of type <Float> and the operand has a value of zero, the result is implementation defined. The implementation must either signal the ZeroDivide exception or provide a continuation value. For all other *numeric representations* the ZeroDivide exception is signaled.

Parameters

`operand` <number> unspecified

Return Values

If the operands conform to integer then <rational>

otherwise the Default Result Type

Errors

`operand` = 0 unless receiver or `operand` are of type <Float>

5.6.2.5 Message: *//* operand

Synopsis

Answer the truncated quotient resulting from dividing the receiver by `operand`. The truncation is towards negative infinity.

Definition: <number>

Answer an integer whose value is the truncated result of dividing the receiver by `operand`, as specified by the ISO/IEC 10967 flooring division operation div^f. Truncation is towards negative infinity. The sign of the result is positive if the receiver and `operand` have the same sign, and negative if the signs are different.

To perform the operation both the receiver and `operand` must be objects with identical *numeric representations*. If they have different representations a conversion to a common *numeric representation* is performed, as specified by the Default Conversion Table, before applying the operation. If the resulting protocol is <integer>, then the result value is defined by the ISO/IEC 10967 operation div^f_I. If the resulting protocol is <Float>, then the result value is defined by the ISO/IEC 10967 div^f_F. Otherwise, the result is consistent with the mathematical definition of the ISO/IEC 10967 operation div^f.

If the operand has a value of zero the ZeroDivide exception is signaled.

Parameters

`operand` <number> unspecified

Return Values

<integer> unspecified

Errors

`operand` = 0 unless receiver or `operand` are of type <Float>

5.6.2.6 Message Refinement: < operand

Synopsis

Answer *true* if the receiver is less than operand. Answer *false* otherwise.

Definition: <magnitude>

Answer *true* if the receiver is less than operand with respect to the ordering defined for them. Answer *false* otherwise.

It is erroneous if the receiver and operand are not *comparable*.

The semantics of the natural ordering must be defined by refinement, which may also restrict the type of operand.

Refinement: <number>

Answer *true* if the `operand` is numerically less than the receiver, as specified by the ISO/IEC 10967 comparison operation *lss*. Answer *false* otherwise.

To perform the operation both the receiver and `operand` must be objects with identical *numeric representations*. If they have different representations a conversion to a common *numeric representation* is performed, as specified by the Default Conversion Table, before applying the operation. If the resulting protocol is <integer>, then the result value is defined by the ISO/IEC 10967 operation lss_I. If the resulting protocol is <Float>, then the result value is defined by the ISO/IEC 10967 lss_F. Otherwise, the result is consistent with the mathematical definition of the ISO/IEC 10967 operation *lss*.

Parameters

`operand` <number> uncaptured

Return Values

<boolean> unspecified

Errors

none

5.6.2.7 Message Refinement: = comparand

Synopsis

Object equivalence test.

Definition: <Object>

This message tests whether the receiver and the `comparand` are equivalent objects at the time the message is processed. Return *true* if the receiver is equivalent to `comparand`. Otherwise return *false*.

The meaning of "equivalent" cannot be precisely defined but the intent is that two objects are considered equivalent if they can be used interchangeably. Conforming protocols may choose to more precisely define the meaning of "equivalent".

The value of

```
receiver = comparand
```

is *true* if and only if the value of

```
comparand = receiver
```

would also be *true*. If the value of

```
receiver = comparand
```

is *true* then the receiver and `comparand` must have *equivalent hash values*. Or more formally:

```
receiver = comparand
receiver hash = comparand hash
```

The equivalence of objects need not be *temporally invariant*. Two independent invocations of #= with the same receiver and `operand` objects may not always yield the same results. Note that a

collection that uses #= to discriminate objects may only reliably store objects whose hash values do not change while the objects are contained in the collection.

Refinement: <number>

Answer *true* if the `operand` is numerically equal to the receiver, as specified by the ISO/IEC 10967 equality operation *eq*. Answer *false* if they are not numerically equal or if `operand` is not a number.

To perform the operation both the receiver and `operand` must be objects with identical *numeric representations*. If they have different representations a conversion to a common *numeric representation* is performed, as specified by the Default Conversion Table, before applying the operation. If the resulting protocol is <integer>, then the result value is defined by the ISO/IEC 10967 operation eq_I. If the resulting protocol is <Float>, then the result value is defined by the ISO/IEC 10967 eq_F. Otherwise, the result is consistent with the mathematical definition of the ISO/IEC 10967 operation *eq*.

Numeric equality is defined by implementation defined conventions regarding round-off error and representation of numbers, hence behavior of this message may differ between platforms.

Parameters

`comparand` <Object> uncaptured

Return Values

<boolean> unspecified

Errors

none

5.6.2.8 Message Refinement: > operand

Synopsis

Answer *true* if the receiver is greater than operand. Answer *false* otherwise.

Definition: <magnitude>

Answer *true* if the receiver is greater than operand with respect to the natural ordering. Answer *false* otherwise.

It is erroneous if the receiver and operand are not *comparable*.

The semantics of the natural ordering must be defined by refinement, which may also restrict the type of operand.

Refinement: <number>

Answer *true* if the `operand` is numerically less than the receiver, as specified by the ISO/IEC 10967 comparison operation *gtr*. Answer *false* otherwise.

To perform the operation both the receiver and `operand` must be objects with identical *numeric representations*. If they have different representations a conversion to a common *numeric representation* is performed, as specified by the Default Conversion Table, before applying the operation. If the resulting protocol is <integer>, then the result value is defined by the ISO/IEC 10967 operation gtr_I. If the resulting protocol is <Float>, then the result value is defined by the ISO/IEC 10967 gtr_F. Otherwise, the result is consistent with the mathematical definition of the ISO/IEC 10967 operation *gtr*.

Parameters

`operand` <number> uncaptured

Return Values

<boolean> unspecified

Errors

none

5.6.2.9 Message: \\ operand

Synopsis

Answer the remainder after integer division of the receiver by the `operand`.

Definition: <number>

Answer the remainder of truncating integer division as specified by the ISO/IEC 10967 remainder operation rem^f. The remainder has the same sign as `operand`. To perform the operation both the receiver and `operand` must be objects with identical *numeric representations*. If they have different representations a conversion to a common *numeric representation* is performed, as specified by the Default Conversion Table, before applying the operation. If the resulting protocol is <integer>, then the result value is defined by the ISO/IEC 10967 operation rem_I. If the resulting protocol is <Float>, then the result value is defined by the ISO/IEC 10967 rem_F. Otherwise, the result is consistent with the mathematical definition of the ISO/IEC 10967 operation *rem*.

The protocol and representation of the return value is defined by the Default Result Type. If the return value conforms to <scaledDecimal> then the scale of the result is at least the scale of the receiver.

Within the limits of representation, the following invariant should hold:

(receiver // `operand`) * `operand` + (receiver \\ `operand`) = receiver

If the result value is outside of the *range* of the common *numeric representation,* the effect of underflow or overflow is implementation defined. If either the receiver or operand is of type <Float> and the operand has a value of zero, the result is implementation defined. The implementation may signal the ZeroDivide exception or provide a continuation value. For all other *numeric representations* the ZeroDivide exception is signaled.

Parameters

`operand` <number> unspecified

Return Values

The Default Result Type

Errors

`operand` = 0 unless receiver or `operand` are of type <Float>

5.6.2.10 Message: abs

Synopsis

Answer the absolute value of the receiver.

Definition: <number>

Return the absolute value of the receiver, as specified by the ISO/IEC 10967 operation *abs*. If the receiver is greater than or equal to zero, answer an object equal to the receiver. Otherwise answer an object which is equal to the negation of the receiver.

Return Values

<RECEIVER> unspecified

Errors

none

5.6.2.11 Message: asFloat

Synopsis

Answer a floating-point number approximating the receiver.

Definition: <number>

Return the nearest floating-point number to the receiver, as specified by the ISO/IEC 10967 *cvt* operation.

If an implementation supports multiple representations for floating point numbers, the result is the representation with the smallest precision that will represent a number with the same number of digits as the receiver, truncating to the maximum precision of the representation with the largest precision.

The effect of underflow or overflow is implementation defined.

Return Values

<Float> unspecified

Errors

None

5.6.2.12 Message: asFloatD

Synopsis

Answer a d precision floating-point number approximating the receiver .

Definition: <number>

Return the nearest floating-point number to the receiver, as specified by the ISO/IEC 10967 *cvt* operation.

Use the object representation for floating point numbers that corresponds to the representation used for numeric literals with the exponent designation 'd'.

The effect of underflow and overflow is implementation defined.

Return Values

<Float> unspecified

Errors

None

5.6.2.13 Message: asFloatE

Synopsis

Answer a floating-point number approximating the receiver.

Definition: <number>

Return the nearest floating-point number to the receiver, as specified by the ISO/IEC 10967 *cvt* operation.

Use the object representation for floating point numbers that corresponds to the representation used for numeric literals with the exponent designation 'e'.

The effect of underflow and overflow is implementation defined.

Return Values

<Float> unspecified

Errors

None

5.6.2.14 Message: asFloatQ

Synopsis

Answer a floating-point number approximating the receiver.

Definition: <number>

Return the nearest floating-point number to the receiver, as specified by the ISO/IEC 10967 *cvt* operation.

Use the object representation for floating point numbers that corresponds to the representation used for numeric literals with the exponent designation 'q'.

The effect of underflow and overflow is implementation defined.

Return Values

<Float> unspecified

Errors

None

5.6.2.15 Message: asFraction

Synopsis

Answer a fraction approximating the receiver.

Definition: <number>

Answer a fraction that reasonably approximates the receiver. If the receiver is an integral value the result may be <integer>.

Return Values

<rational> unspecified

Errors

none

5.6.2.16 Message: asInteger

Synopsis

Answer an integer approximating the receiver.

Definition: <number>

Answer the result of sending `#rounded` to the receiver.

Return Values

<integer> unspecified

Errors

none

5.6.2.17 Message: asScaledDecimal: scale

Synopsis

Answer a scaled decimal number, with a fractional precision of `scale`, approximating the receiver.

Definition: <number>

This is a conversion message. Answer a scaled decimal number, with a fractional precision of `scale`, which minimizes the difference between the answered value and the receiver.

The effect of underflow and overflow is implementation defined.

Return Values

<scaledDecimal> unspecified

Errors

None

5.6.2.18 Message: ceiling

Synopsis

Answer the smallest integer greater than or equal to the receiver.

Definition: <number>

Answer the smallest integer greater than or equal to the receiver.

Return Values

<integer> unspecified

Errors

none

5.6.2.19 Message: floor

Synopsis

Answer the largest integer less than or equal to the receiver.

Definition: <number>

Answer the largest integer less than or equal to the receiver.

Return Values

<integer> unspecified

Errors

none

5.6.2.20 Message: fractionPart

Synopsis

Answer the fractional part of the receiver.

Definition: <number>

Return an object conforming to the protocol of the receiver that is equal to the fractional part of the receiver. Within the limits of representation, the following invariants should hold:

receiver integerPart + receiver fractionPart = receiver

receiver \\1 = receiver fractionPart

Return Values

<RECEIVER> unspecified

Errors

none

5.6.2.21 Message: integerPart

Synopsis

Answer the integer part of the receiver.

Definition: <number>

Return an object that is equal to the integer part of the receiver. If the receiver is type <Fraction> return an object conforming to <integer>. Otherwise return an object conforming to the protocol of the receiver.

Return Values

receiver	result
<rational>	<rational>
<scaledDecimal>	<scaledDecimal>
<Float>	<Float>

Rationale

The return value is not restricted to <integer> to avoid unnecessary mixed mode arithmetic.

Errors

none

5.6.2.22 Message: negated

Synopsis

Answer the negation of the receiver.

Definition: <number>

Answer an object conforming to the receiver's protocol that is equal to the negation of the receiver (equal in magnitude to the receiver but opposite in sign), as specified by the ISO/IEC 10967 *neg* operation.

Return Values

<RECEIVER> unspecified

Errors

none

5.6.2.23 Message: negative

Synopsis

Answer *true* if the receiver is negative.

Definition: <number>

Answer *true* if the receiver is negative. Answer *false* otherwise.

Return Values

<boolean> unspecified

Errors

none

5.6.2.24 Message: positive

Synopsis

Answer *true* if the receiver is positive or zero.

Definition: <number>

Answer *true* if the receiver is positive or zero. Answer *false* otherwise.

Return Values

<boolean> unspecified

Errors

none

5.6.2.25 Message Refinement: printString

Synopsis

Return a string that describes the receiver.

Definition: <Object>

A string consisting of a sequence of characters that describe the receiver are returned as the result.

The exact sequence of characters that describe an object are implementation defined.

Refinement: <number>

Answer a string that is a valid literal representation that approximates the numeric value of the receiver.

Return Values

<readableString> unspecified

Errors

none

5.6.2.26 Message: quo: operand

Synopsis

Answer the truncated integer quotient resulting from dividing the receiver by `operand`. Truncation is towards zero.

Definition: <number>

Answer a number whose value is the result of dividing the receiver by `operand`, as specified by the ISO/IEC 10967 flooring division operation *div* To perform the operation both the receiver and `operand` must be objects with identical *numeric representations*. If they have different representations a conversion to a common *numeric representation* is performed, as specified by the Default Conversion Table, before applying the operation. If the resulting protocol is <integer>, then the result value is a <rational> with the receiver as the numerator and the `operand` as the denominator. If the resulting protocol is <Float>, then the result value is defined by the ISO/IEC 10967 div_F. Otherwise, the result is consistent with the mathematical definition of the ISO/IEC 10967 operation *div*.

The protocol and representation of the return value are defined by the Default Result Type. If the return value conforms to <scaledDecimal> then the scale of the result is at least the scale of the receiver after conversion if necessary.

If the result value is outside of the *range* of the common *numeric representation,* the effect of underflow or overflow is implementation defined. If either the receiver or operand are of type <Float> and the operand has a value of zero, the result is implementation defined. The implementation must either signal the ZeroDivide exception or provide a continuation value. For all other *numeric representations* the ZeroDivide exception is signaled.

Parameters

`operand` <number> unspecified

Return Values

<integer> unspecified

Errors

`operand` = 0 unless receiver or `operand` are of type <Float>

5.6.2.27 Message: raisedTo: operand

Synopsis

Answer the receiver raised to the power `operand`.

Definition: <number>

If `operand` conforms to <integer>, answer the result of sending `#raisedToInteger:` with argument `operand` to the receiver.

Otherwise answer

```
(receiver asFloat ln * operand) exp.
```

It is erroneous if the receiver equals zero and the `operand` is less than or equal to zero, or if the receiver is less than zero. The effect of underflow and overflow is implementation defined.

If the *numeric representation* of the result has does not have *unbounded precision,* the effect of underflow or overflow is implementation defined.

Parameters

`operand` <number> uncaptured

Return Values

<number> unspecified

Errors

receiver = 0 and `operand` <= 0

receiver < 0

5.6.2.28 Message: raisedToInteger: operand

Synopsis

Answer the receiver raised to the power `operand`.

Definition: <number>

Answer the receiver raised to the power `operand`, which must be a whole number. If the `operand` is a whole number greater than or equal to zero, then the result is the receiver raised to the power `operand`. If `operand` is a negative whole number then the result is equivalent to the reciprocal of the absolute value of the receiver raised to the power `operand`.

It is erroneous if the `operand` does not conform to the protocol <integer>. If the *numeric representation* of the result has does not have *unbounded precision,* the effect of underflow or overflow is implementation defined.

Parameters

`operand` <integer> uncaptured

Return Values

<RECEIVER> unspecified

Errors

Receiver is not an integer.

5.6.2.29 Message: reciprocal

Synopsis

Answer the reciprocal of the receiver.

Definition: <number>

Answer the reciprocal of the receiver, which is equal to the result of the operation (1/receiver). Signal a ZeroDivide exception if the receiver is equal to zero.

Return Values

receiver	result
<integer>	<rational>
<Fraction>	<rational>
<scaledDecimal>	<scaledDecimal>
<Float>	<Float>

Errors

receiver = 0

5.6.2.30 Message: rem: operand

Synopsis

Answer the remainder after integer division of the receiver by the `operand`.

Definition: <number>

Answer the remainder with respect to integer division, as specified by the ISO/IEC 10967 remainder operation *rem*. The sign of the remainder is the same sign as the receiver. Within the limits of representation, the following invariant should hold:

`(receiver quo: operand)*operand + receiver rem: operand) = receiver`

To perform the operation both the receiver and `operand` must be objects with identical *numeric representations*. If they have different representations a conversion to a common *numeric representation* is performed, as specified by the Default Conversion Table,

The protocol and representation of the return value is defined by the Default Result Type. If the return value conforms to <scaledDecimal> then the scale of the result is at least the scale of the receiver after conversion if necessary. If either the receiver or operand are of type <Float> and the operand has a value of zero, the result is implementation defined. The implementation may signal the ZeroDivide exception or provide a continuation value. For all other *numeric representations* the ZeroDivide exception is signaled. If the result value is outside of the *range* of the common *numeric representation,* the effect of underflow or overflow is implementation defined.

Parameters

`operand` <number> unspecified

Return Values

<number> unspecified

Errors

`operand` = 0 unless receiver or `operand` are of type <Float>

5.6.2.31 Message: rounded

Synopsis

Answer the integer nearest the receiver.

Definition: <number>

Answer the integer nearest the receiver according to the following property:

N rounded = the nearest integer I = N + (N sign * (1/2)) truncated towards zero.

For example, 0.5 rounded = 1 and -0.5 rounded = -1.

Return Values

<integer> unspecified

Errors

None

5.6.2.32 Message: roundTo: factor

Synopsis

Answer the number nearest the receiver that is a multiple of `factor`.

Definition: <number>

Answer the number nearest the receiver that is a multiple of `factor`. The result conforms to either the receiver's or `operand`'s protocol, according to the Default Conversion Table.

The result is undefined if `factor` equals zero. If the *numeric representation* of the result has does not have *unbounded precision,* the effect of underflow or overflow is implementation defined.

Parameters

`factor` <number> uncaptured

Return Values

The Default Result Type

Errors

None

5.6.2.33 Message: sign

Synopsis

Answer the sign of the receiver.

Definition: <number>

Answer 1 if the receiver is positive, 0 if the receiver equals 0, and -1 if it is negative, as specified by the ISO/IEC 10967 operation *sign*.

Return Values

<integer> unspecified

Errors

none

5.6.2.34 Message: sqrt

Synopsis

Answer the positive square root of the receiver.

Definition: <number>

Answer a number equal to the positive square root of the receiver as specified by the ISO/IEC 10967 remainder operation *sqrt*. If the receiver's protocol is <integer>, then the result value is defined by the ISO/IEC 10967 operation $sqrt_I$. If the receiver's protocol is <Float>, then the result value is defined by the ISO/IEC 10967 $sqrt_F$. Otherwise, the result is consistent with the mathematical definition of the ISO/IEC 10967 operation *sqrt*.

The result is undefined if the receiver is less than zero.

Return Values

<number> unspecified

Errors

none

5.6.2.35 Message: squared

Synopsis

Answer the receiver squared.

Definition: <number>

Answer a number that is the receiver multiplied by itself. The answer must conform to the same protocol as the receiver.

Return Values

<RECEIVER> unspecified

Errors

none

5.6.2.36 Message: strictlyPositive

Synopsis

Answer *true* if the receiver is greater than zero.

Definition: <number>

Answer *true* if the receiver is greater than zero.

Return Values

<boolean> unspecified

Errors

none

5.6.2.37 Message: to: stop

Synopsis

Answer an object conforming to <interval> which represents an arithmetic progression from the receiver to `stop` in increments of 1.

Definition: <number>

Answer an interval which represents an arithmetic progression from the receiver to `stop`, using the increment 1 to compute each successive element. The elements conform to the receiver's protocol. Note that `stop` may not be the last element in the sequence, which is given by the formula

```
receiver + ((stop - receiver) // 1)
```

The interval answered will be empty if the receiver is greater than `stop`.

Parameters

`stop`	<number>	unspecified

Return Values

<Interval>	unspecified

Errors

none

5.6.2.38 Message: to: stop by: step

Synopsis

Answer an interval which represents an arithmetic progression from receiver to `stop` in increments of `step`.

Definition: <number>

Answer an interval which represents an arithmetic progression from the receiver to `stop`, using the increment `step` to compute each successive element. The value of `step` can be positive or negative, but it must be non-zero. The elements conform to either the receiver's or `step`'s protocol, according to the Default Conversion Table.

Note that `stop` may not be the last element in the sequence, which is given by the formula

```
(((stop - receiver) // step) * step) + receiver
```

The interval answered will be empty if:

1. receiver < `stop`, and `step` < 0.
2. receiver > `stop`, and `step` > 0.

Parameters

`stop`	<number>	unspecified
`step`	<number>	unspecified

Return Values

<Interval>	unspecified

Errors

`step` = 0

5.6.2.39 Message: to: stop by: step do: operation

Synopsis

Evaluate `operation` for each element of an interval which represents an arithmetic progression from the receiver to `stop` in increments of `step`.

Definition: <number>

Evaluate `operation` for each element of an interval starting at the receiver and stopping at `stop` where each element is `step` greater than the previous. The value of `step` can be positive or negative, but it must be non-zero. The elements must all conform to either the receiver's or `step`'s protocol, according to the Default Conversion Table.

Note that `stop` is not necessarily an element in the sequence, which is given by the formula

```
(((stop - receiver) // step) * step) + receiver
```

No evaluation takes place if:

1. receiver < `stop`, and `step` < 0.
2. receiver > `stop`, and `step` > 0.

Implementations are not required to actually create the interval described by the receiver, `stop` and `step`. Implementations may restrict the definition of this message to specific classes.

Parameters

`stop`	<number>	unspecified
`step`	<number>	unspecified
`operation`	<monadicBlock>	unspecified

Return Values

UNSPECIFIED

Errors

`step` = 0

5.6.2.40 Message: to: stop do: operation

Synopsis

Evaluate `operation` for each element of an interval which represents an arithmetic progression from receiver to `stop` in increments of 1.

Definition: <number>

Evaluate `operation` for each element of an interval starting at the receiver and stopping at `stop` where each element is 1 greater than the previous. The elements must all conform to the receiver's protocol according to the Default Conversion Table.

Note that `stop` may not be the last element in the sequence, which is given by the formula

```
receiver + ((stop - receiver) // 1)
```

No evaluation takes place if the receiver is greater than `stop`.

Implementations are not required to actually create the interval described by the receiver and `stop`.

Parameters

`stop`	<number>	unspecified
`operation`	<monadicBlock>	unspecified

Return Values

UNSPECIFIED

Errors

none

5.6.2.41 Message: truncated

Synopsis

Answer an integer equal to the receiver truncated towards zero.

Definition: <number>

As specified by the ISO/IEC 10967 truncation operation *trunc*. If the receiver is positive, answer the largest integer less than or equal to the receiver. If it is negative, answer the smallest integer greater than or equal to the receiver.

Return Values

<integer> unspecified

Errors

none

5.6.2.42 Message: truncateTo: factor

Synopsis

Answer the number nearest the receiver truncated towards zero which is a multiple of `factor`.

Definition: <number>

If the receiver is positive, answer the largest number less than or equal to the receiver which is a multiple of `factor`. If it is negative, answer the smallest number greater than or equal to the receiver which is a multiple of `factor`.

The type of the return value depends on the type of the receiver and factor, as indicated by the Default Conversion Table.

Parameters

`factor` <number> uncaptured

Return Values

The Default Result Type

Errors

none

5.6.3 Protocol: <rational>

Conforms To

<number>

Description

Rational numbers may be either integers or fractions. An integer is logically a fraction whose denominator is one. This protocol is necessary because some integer and most fraction operations can produce results that may be either an integer or a fraction.

Messages

denominator
numerator

5.6.3.1 Message: denominator

Synopsis

Answer the denominator of the receiver.

Definition: <rational>

Treating the receiver as a fraction, answer the lowest common denominator of the receiver.

Return Values

<integer> unspecified

Errors

none

5.6.3.2 Message: numerator

Synopsis

Answer the numerator of the receiver.

Definition: <rational>

Treating the receiver as a fraction reduced to its lowest common denominator, answer the integer numerator.

Return Values

<integer> unspecified

Errors

none

5.6.4 Protocol: <Fraction>

Conforms To

<rational>

Description

An exact representation for rational numbers. It is unspecific whether the rational number are maintain in a reduced form but messages that reveal the numerator and denominator answer values as if the fraction was reduced.

Messages

denominator
numerator
printString

5.6.4.1 Message Refinement: denominator

Synopsis

Answer the denominator of the receiver.

Definition: <rational>

Treating the receiver as a fraction, answer the lowest common denominator of the recevier.

Refinement: <Fraction>

Answer the integer smallest integer denominator of the receiver.

Return Values

<integer>unspecified

Errors

none

5.6.4.2 Message Refinement: numerator

Synopsis

Answer the numerator of the receiver.

Definition: <rational>

Treating the receiver as a fraction, answer the integer numerator.

Refinement: <Fraction>

Answer the integer numerator of the receiver reduced to its lowest denominator.

Return Values

<integer>unspecified

Errors

none

5.6.4.3 Message Refinement: printString

Definition: <Object>

A string consisting of a sequence of characters that describe the receiver are returned as the result.

The exact sequence of characters that describe an object are implementation defined.

Refinement: <number>

Answer a string that is a valid literal representation equal to the receiver.

Refinement: <Fraction>

Answer a string consisting of the numerator and denominator for a reduced fraction, equivalent to the receiver . The numerator and denominator are separated by the character '/' as follows:

numerator/denominator

Return Values

<readableString> unspecified

Errors

none

5.6.5 Protocol: <integer>

Conforms To

<rational>

Description

Represents an abstraction for integer numbers whose value is exact. Representations must provide *unbounded precision* and *range*, hence the ISO/IEC 10967 integer type parameter *bounded* is bound to false.

Messages:

allMask:
anyMask:
asScaledDecimal:
bitAnd:
bitAt:
bitAt:put:
bitOr:
bitShift:
bitXor:
even
factorial
gcd:
highBit
lcm:
noMask:
odd
printStringRadix:
printOn:base:showRadix:

5.6.5.1 Message: allMask: mask

Synopsis

Answer *true* if all of the bits that are 1 in the binary representation of `mask` are 1 in the binary representation of the receiver. Answer *false* otherwise.

Definition: <integer>

Answer *true* if all of the bits that are 1 in the binary representation of `mask` are 1 in the binary representation of the receiver. Answer *false* otherwise. If the receiver has fewer bits than the operand, the receiver is treated as if it were extended on the left with zeros to the length of the operand.

The result is undefined if either the receiver or the `operand` is a negative integer.

Parameters

`mask` <integer> uncaptured

Return Values

<boolean> unspecified

Errors

none

5.6.5.2 Message: anyMask: mask

Synopsis

Answer *true* if any of the bits that are 1 in the binary representation of `mask` are 1 in the binary representation of the receiver. Answer *false* otherwise.

Definition: <integer>

Answer *true* if any of the bits that are 1 in the binary representation of `mask` are 1 in the binary representation of the receiver. Answer *false* otherwise. If the receiver has fewer bits than the operand, the receiver is treated as if it were extended on the left with zeros to the length of the operand.

Result is undefined if either the receiver or the operand is a negative integer.

Parameters

`mask` <integer> uncaptured

Return Values

<boolean> unspecified

Errors

none

5.6.5.3 Message Refinement: asScaledDecimal: scale

Synopsis

Answer a scaled decimal number, with a fractional precision of `scale`, approximating the receiver.

Definition: <number>

This is a conversion message. Answer a scaled decimal number, with a fractional precision of `scale`, which minimizes the difference between the answered value and the receiver.

The effect of underflow and overflow is implementation defined.

Refinement: <integer>

The number of significant digits of the answer is the same as the number of decimal digits in the receiver. The scale of the answer is 0.

It is an error if the receiver cannot be represented within the maximum precision of the <scaledDecimal> implementation.

Return Values

<scaledDecimal> unspecified

Errors

scaled decimal overflow

5.6.5.4 Message: bitAnd: operand

Synopsis

Answer the bit-wise logical and of the receiver and the `operand`.

Definition: <integer>

Answer the result of the bit-wise logical and of the binary representation of the receiver and the binary representation of `operand`. The shorter of the receiver or the `operand` is extended on the left with zeros to the length of the longer of the two.

The result is undefined if either the receiver or the operand is a negative integer.

Parameters

`operand` <integer> uncaptured

Return Values

<integer> unspecified

Errors

none

5.6.5.5 Message: bitAt: index

Synopsis

Answer the value of the bit at `index` in the binary representation of the receiver.

Definition: <integer>

Answer the value of the bit at `index` in the binary representation of the receiver. Answer an integer value of 0 or 1, depending upon the value of the bit at position `index` in the binary representation

of the receiver. The least significant bit of the receiver is designated as bit 1, with indices increasing to the left.

The result is undefined if either the receiver is negative. It is erroneous if `index` is less that or equal to zero.

Parameters

`index` <integer> uncaptured

Return Values

<integer> unspecified

Errors

`index` less than or equal to zero

5.6.5.6 Message: bitAt: index put: value

Synopsis

Set the value of the bit at `index` in the binary representation of the receiver.

Definition: <integer>

Return an integer whose binary representation is identical to the receiver with the exception that the value of the bit at position `index` is equal to the low order bit of `value`.

The least significant bit of the receiver is designated as position 1, with indices increasing to the left.

The result is undefined if either the receiver or `value` is a negative integer. It is erroneous if `index` is less that or equal to zero.

Parameters

`index` <integer> uncaptured

Return Values

<integer> unspecified

Errors

`index` less than or equal to zero

5.6.5.7 Message: bitOr: operand

Synopsis

Answer the logical or of the receiver and `operand`.

Definition: <integer>

Answer the result of bit-wise logical or the binary representation of the receiver and the binary representation of `operand`. The shorter of the receiver or the `operand` is extended on the left with zeros to the length of the longer of the two.

The result is undefined if either the receiver or the operand is a negative integer.

Parameters

`operand` <integer> uncaptured

Return Values

<integer> unspecified

Errors

none

5.6.5.8 Message: bitShift: shift

Synopsis

Answer the result of logically bit-wise shifting the binary representation of the receiver by `shift` bits.

Definition: <integer>

If `shift` is positive, the receiver is shifted left and zeros (0) are shifted in on the right. If `shift` is negative, the receiver is shifted right and low order bits are discarded.

The result is undefined if either the receiver is negative.

Parameters

`shift` <integer> uncaptured

Return Values

<integer> unspecified

Errors

none

5.6.5.9 Message: bitXor: operand

Synopsis

Answer bit-wise exclusive or of the receiver and the `operand`.

Definition: <integer>

Answer the result of the bit-wise exclusive or of the binary representation of the receiver and the binary representation of `operand`. The shorter of the receiver or the `operand` is extended on the left with zeros to the length of the longer of the two.

The result is undefined if either the receiver or the operand is a negative integer.

Parameters

`operand` <integer> uncaptured

Return Values

<integer> unspecified

Errors

none

5.6.5.10 Message: even

Synopsis

Answer *true* if the receiver is even.

Definition: <integer>

Answer *true* if the receiver is divisible by 2 with no remainder.

Return Values

<boolean> unspecified

Errors

none

5.6.5.11 Message: factorial

Synopsis

Answer the factorial of the receiver.

Definition: <integer>

Answer the product of all numbers between the receiver and 1 inclusive. The result is undefined if the receiver is negative.

Return Values

<integer> unspecified

Errors

none

5.6.5.12 Message: gcd: operand

Synopsis

Answer the greatest common divisor of the receiver and `operand`.

Definition: <integer>

Answer the largest non-negative integer that divides both the receiver and `operand` with no remainder. Answer 0 if the receiver and `operand` are zero.

Parameters

`operand`	<integer>	uncaptured

Return Values

<integer> unspecified

Errors

none

5.6.5.13 Message: highBit

Synopsis

Answer the index of the most significant non-zero bit in the binary representation of the receiver.

Definition: <integer>

Answer the index of the most significant non-zero bit in the binary representation of the receiver. Answer 0 if the receiver is 0. The index of the least significant bit of the receiver is 1, with indices increasing to the left.

The result is undefined if the receiver is negative.

Return Values

<integer> unspecified

Errors

none

5.6.5.14 Message: lcm: operand

Synopsis

Answer the least common multiple of the receiver and `operand`.

Definition: <integer>

Answer the smallest non-negative integer which is evenly divided by both the receiver and `operand`. Answer 0 if the receiver and `operand` are zero.

Parameters

`operand`	<integer>	uncaptured

Return Values

<integer> unspecified

Errors

none

5.6.5.15 Message: noMask: mask

Synopsis

Answer *true* if none of the bits that are 1 in the binary representation of `mask` are 1 in the binary representation of the receiver. Answer *false* otherwise.

Definition: <integer>

Answer *true* if none of the bits that are 1 in the binary representation of `mask` are 1 in the binary representation of the receiver. Answer *false* otherwise. If the receiver has fewer bits than the operand, the receiver is treated as if it were extended on the left with zeros to the length of the operand.

The result is undefined if either the receiver or the `operand` is a negative integer.

Parameters

`mask` <integer> uncaptured

Return Values

<boolean> unspecified

Errors

none

5.6.5.16 Message: odd

Synopsis

Answer *true* if the receiver is odd.

Definition: <integer>

Answer true if the receiver is divisible by two (2) with remainder one (1).

Return Values

<boolean> unspecified

Errors

none

5.6.5.17 Message: printStringRadix: base

Synopsis

Answer a string which represents the receiver in radix `base`.

Definition: <integer>

Return a string containing a sequence of characters that represents the numeric value of the receiver in the radix specified by the argument. The sequence of characters must be recognizable using the radixDigits production of the Smalltalk Lexical Grammar as if the numeric value of the radixSpecifier was `base`. If the receiver is negative, a minus sign ('-') is prepended to the sequence of characters. The result is undefined if `base` is less than two or greater than 36.

Parameters

`base` <integer> uncaptured

Return Values

<readableString> unspecified

Errors

none

5.6.5.18 Message: printOn: output base: base showRadix: flag

Synopsis

Write a sequence of characters that describes the receiver in radix `base` with optional radix specifier.

Definition: <integer>

Write to output a sequence of characters that describes the receiver, starting at output's current position. If the parameter flag is true, produce a sequence of characters that are recognizable using the radixInteger production of the Smalltalk Lexical Grammar. If the flag is false, then the sequence of characters must be recognizable using the radixDigits production as if the numeric value of the radixSpecifier was `base`. If the receiver is negative, a minus sign ('-') is prepended to the sequence of characters. The result is undefined if `base` is less than two or greater than 36.

Parameters

`output`	<puttableStream>	uncaptured
`base`	<integer>	uncaptured
`flag`	<boolean>	uncaptured

Return Values

UNSPECIFIED

Errors

none

5.6.6 Protocol: <scaledDecimal>

Conforms To

<number>

Description

Provides a numeric representation of fixed point decimal numbers. The representation must be able to accurately represent decimal fractions. The standard recommends that the implementation of this protocol support unbounded precision, with no limit to the number of digits before and after the decimal point. If a bounded implementation is provided, then any operation which exceeds the bounds has an implementation-specified result.

Messages:

scale

5.6.6.1 Message: scale

Synopsis

Answer a integer which represents the total number of digits used to represent the fraction part of the receiver, including trailing zeroes.

Definition: <scaledDecimal>

Answer a integer which represents the total number of digits used to represent the fraction part of the receiver, including trailing zeroes.

Return Values

<integer> unspecified

Errors

none

5.6.7 Protocol: <Float>

Conforms To

<number>

Description

Represents a floating point representation for real numbers, whose value may be approximate. Provides protocol for performing trigonometry, exponentiation, and conversion on numerical quantities.

Operations can produce results that are outside the set of representable numbers, or that are mathematically undefined. It is implementation defined whether errors are raised when results are not representable or if unrepresentable results are wrapped in implementation-defined continuation values or their equivalent. The effect of underflow and overflow is therefore implementation defined. It is erroneous if the result of an operation is mathematically undefined.

Messages:

=
arcCos
arcSin
arcTan
cos
degreesToRadians
exp
floorLog:
ln
log:
printString
radiansToDegrees
sin
tan

5.6.7.1 Message Refinement: = comparand

Synopsis

Object equivalence test.

Definition: <Object>

This message tests whether the receiver and the `comparand` are equivalent objects at the time the message is processed. Return *true* if the receiver is equivalent to `comparand`. Otherwise return *false*.

The meaning of "equivalent" cannot be precisely defined but the intent is that two objects are considered equivalent if they can be used interchangeably. Conforming protocols may choose to more precisely define the meaning of "equivalent".

The value of

```
receiver = comparand
```

is *true* if and only if the value of

```
comparand = receiver
```

would also be *true*. If the value of

```
receiver = comparand
```

is *true* then the receiver and `comparand` must have *equivalent hash values*. Or more formally:

```
receiver = comparand
receiver hash = comparand hash
```

The equivalence of objects need not be *temporally invariant*. Two independent invocations of #= with the same receiver and `operand` objects may not always yield the same results. Note that a collection that uses #= to discriminate objects may only reliably store objects whose hash values do not change while the objects are contained in the collection.

Refinement: <number>

Answer *true* if the `operand` is numerically equal to the receiver, as specified by the ISO/IEC 10967 equality operation *eq*. Answer *false* if they are not numerically equal or if `operand` is not a number.

To perform the operation both the receiver and `operand` must be objects with identical *numeric representations*. If they have different representations a conversion to a common *numeric representation* is performed, as specified by the Default Conversion Table, before applying the operation. If the resulting protocol is <integer>, then the result value is defined by the ISO/IEC 10967 operation eq_I. If the resulting protocol is <Float>, then the result value is defined by the ISO/IEC 10967 eq_F. Otherwise, the result is consistent with the mathematical definition of the ISO/IEC 10967 operation *eq*.

Numeric equality is defined by implementation defined conventions regarding round-off error and representation of numbers, hence behavior of this message may differ between platforms.

Refinement: <Float>

Answer *true* if the `operand` is a number which represents the same floating point number as the receiver, as specified by the ISO/IEC 10967 operation eq_f. If the `comparand` and the receiver do not conform to the same protocol, they are converted according to the Default Conversion Table.

Parameters

`comparand`	<Object>	uncaptured

Return Values

<boolean>	unspecified

Errors

none

5.6.7.2 Message: arcCos

Synopsis

Answer the inverse cosine of the receiver in radians.

Definition: <Float>

Answer the inverse cosine of the receiver in radians, as specified by the ISO/IEC 10967 trigonometric operation $arccos_f$. Within the limits of precision, the following invariant holds:

```
receiver arcCos cos = receiver
```

It is erroneous if the absolute value of the receiver is greater than 1.

Return Values

<Float> unspecified

Errors

|receiver| > 1

5.6.7.3 Message: arcSin

Synopsis

Answer the inverse sine of the receiver in radians.

Definition: <Float>

Answer the inverse sine of the receiver in radians, as specified by the ISO/IEC 10967 trigonometric operation $arcsin_f$. Within the limits of precision, the following invariant holds:

```
receiver arcSin sin = receiver
```

It is erroneous if the absolute value of the receiver is greater than 1.

Return Values

<Float> unspecified

Errors

|receiver| > 1

5.6.7.4 Message: arcTan

Synopsis

Answer the inverse tangent of the receiver in radians.

Definition: <Float>

Answer the inverse tangent of the receiver in radians, as specified by the ISO/IEC 10967 trigonometric operation $arctan_f$. Within the limits of precision, the following invariant holds:

```
receiver arcTan tan = receiver
```

Return Values

<Float> unspecified

Errors

none

5.6.7.5 Message: cos

Synopsis

Answer the cosine of the receiver in radians.

Definition: <Float>

Answer a <Float> equal to the cosine of the receiver in radians, as specified by the ISO/IEC 10967 trigonometric operation cos_f.

The effect of underflow is implementation defined.

Return Values

<Float> unspecified

Errors

none

5.6.7.6 Message: degreesToRadians

Synopsis

Answer the receiver converted from degrees to radians.

Definition: <Float>

Answer a floating-point number representing the receiver converted from degrees to radians. The result is equivalent to multiplying the receiver by (Pi / 180).

Return Values

<Float> unspecified

Errors

none

5.6.7.7 Message: exp

Synopsis

Answer the natural exponential of the receiver. This is the inverse of `#ln`.

Definition: <Float>

Answer a floating-point number representing the irrational number e (= 2.718281...) raised to the power of the receiver, as specified by the ISO/IEC 10967 operation exp_f. This is the inverse of the `#ln` message.

The effect of underflow and overflow is implementation defined.

Return Values

<Float> unspecified

Errors

none

5.6.7.8 Message: floorLog: operand

Synopsis

Answer the largest integer less than or equal to the logarithm to the base `operand` of the receiver.

Definition: <Float>

Answer the largest integer less than or equal to the power to which the `operand` must be raised to obtain the receiver (that is, the logarithm base `operand` of the receiver).

The result is undefined if the receiver is less than or equal to zero, or if the `operand` is less than or equal to 1.

Parameters

`operand` <number> uncaptured

Return Values

<integer> unspecified

Errors

none

5.6.7.9 Message: ln

Synopsis

Answer the natural logarithm of the receiver.

Definition: <Float>

Answer the natural logarithm of the receiver, as specified by the ISO/IEC 10967 operation ln_f., which is a floating-point number representing the power to which the irrational number e (= 2.718281...) must be raised to obtain the receiver. This is the inverse of the `#exp` message.

The result is undefined if the receiver is less than or equal to zero.

Return Values

<Float> unspecified

Errors

none

5.6.7.10 Message: log: operand

Synopsis

Answer the logarithm to the base `operand` of the receiver.

Definition: <Float>

Answer the logarithm to the base `operand` of the receiver, as specified by the ISO/IEC 10967 operation log_{ff}, which is a floating-point number representing the power to which `operand` must be raised to obtain the receiver. The receiver must be positive, and `operand` must be greater than one. This is the inverse of the `#raisedTo:` message.

The result is undefined if `operand` equals 1, if `operand` is less than or equal to zero, or if the receiver is less than or equal to zero. The effect of underflow and overflow is implementation defined.

Parameters

`operand` <number> uncaptured

Return Values

<Float> unspecified

Errors

none

5.6.7.11 Message Refinement: printString

Synopsis

Return a string that describes the receiver.

Definition: <Object>

A string consisting of a sequence of characters that describe the receiver are returned as the result.

The exact sequence of characters that describe an object are implementation defined.

Refinement: <number>

Answer a string that is a valid literal representation that approximates the numeric value of the receiver.

Refinement: <Float>

Answer a string which is a valid Smalltalk literal representation approximately equal to the receiver. An exponent literal form is produced if the value of the exponent is greater than the precision of the receiver.

Return Values

<readableString> unspecified

Errors

none

5.6.7.12 Message: radiansToDegrees

Synopsis

Answer the receiver converted from radians to degrees.

Definition: <Float>

Answer a floating-point number representing the receiver converted from radians to degrees. The result is equivalent to multiplying the receiver by (180 / Pi).

Return Values

<Float> unspecified

Errors

none

5.6.7.13 Message: sin

Synopsis

Answer the sine of the receiver.

Definition: <Float>

Answer a floating-point number equal to the sine of the receiver in radians, as specified by the ISO/IEC 10967 trigonometric operation sin_F.

The effect of underflow is implementation defined.

Return Values

<Float> unspecified

Errors

none

5.6.7.14 Message: tan

Synopsis

Answer the tangent of the receiver.

Definition: <Float>

Answer a floating-point number equal to the tangent of the receiver in radians, as specified by the ISO/IEC 10967 trigonometric root operation tan_f.

The effect of underflow and overflow is implementation defined.

Return Values

<Float> unspecified

Errors

none

5.6.8 Protocol: <floatCharacterization>

Conforms To

<Object>

Description

Objects supporting this protocol characterize a floating point representation for real numbers. These characterizations are required by ISO/IEC 10967 for each precision of floating point numbers provided by an implementation.

Standard Globals

`Float`	Conforms to the protocol <floatCharacterization>. Its language element type is implementation defined. The value of this global is equivalent to the value of one of the globals: `FloatE`, `FloatE`, or `FloatE`.
`FloatE`	Conforms to the protocol <floatCharacterization>. Its language element type is implementation defined. This global characterizes the floating point representation corresponding to the 'e' floating point literal syntax.
`FloatD`	Conforms to the protocol <floatCharacterization>. Its language element type is implementation defined. This global characterizes the floating point representation corresponding to the 'd' floating point literal syntax.
`FloatQ`	Conforms to the protocol <floatCharacterization>. Its language element type is implementation defined. This global characterizes the floating point representation corresponding to the 'q' floating point literal syntax.

Messages

denormalized
e
emax
emin
epsilon
fmax
fmin
fminDenormalized
fminNormalized
pi
precision
radix

5.6.8.1 Message: denormalized

Synopsis

Indication of whether the characterized floating point object representation allows denormalized values.

Definition: <floatCharacterization>

Report a boolean indicating whether the characterized floating point object representation contains denormalized values. This satisfies the ISO/IEC 10967 floating point characterization requirement *denorm*.

Return Values

<boolean> unspecified

Errors

none

5.6.8.2 Message: e

Synopsis

The closest floating point approximation of the irrational number e.

Definition: <floatCharacterization>

Return the closest floating point approximation of the irrational number e for the characterized floating point object representation.

Return Values

<Float> unspecified

Errors

none

5.6.8.3 Message: emax

Synopsis

The largest exponent of the characterized floating point object representation.

Definition: <floatCharacterization>

Report the largest exponent allowed by the characterized floating point object representation, providing the upper bound of the range of representable floating point numbers. This satisfies the ISO/IEC 10967 floating point characterization requirement *emax*.

Return Values

<integer> unspecified

Errors

none

5.6.8.4 Message: emin

Synopsis

The smallest exponent of the characterized floating point object representation.

Definition: <floatCharacterization>

Report the smallest exponent allowed by the characterized floating point object representation, providing the lower bound of the range of representable floating point numbers. This satisfies the ISO/IEC 10967 floating point characterization requirement *emin*.

Return Values

<integer> unspecified

Errors

none

5.6.8.5 Message: epsilon

Synopsis

The maximum relative spacing in the characterized floating point object representation.

Definition: <floatCharacterization>

Report the maximum relative spacing in the characterized floating point object representation, satisfying the ISO/IEC 10967 floating point characterization requirement *epsilon*. The return value is equal to

```
self radix raisedTo: (1 - self precision)
```

Return Values

<Float> unspecified

Errors

none

5.6.8.6 Message: fmax

Synopsis

The largest value allowed by the characterized floating point object representation.

Definition: <floatCharacterization>

Report the largest value allowed by the characterized floating point object representation. This satisfies the ISO/IEC 10967 floating point characterization requirement *fmax*, and is equal to

```
(1 - (self radix raisedTo: self precision negated)) * self radix
raisedTo: self emax
```

Return Values

<Float> unspecified

Errors

none

5.6.8.7 Message: fmin

Synopsis

The minimum value allowed by the characterized floating point object representation.

Definition: <floatCharacterization>

Report the minimum value allowed by the characterized floating point object representation. This satisfies the ISO/IEC 10967 floating point characterization requirement *fmin*. If the described representation contains normalized values, then the result is equal to the result of sending `#fminNormalized` to the receiver, otherwise the result is equal to the result of sending `#fminDenormalized` to the receiver.

Return Values

<Float> unspecified

Errors

none

5.6.8.8 Message: fminDenormalized

Synopsis

The minimum denormalized value allowed by the characterized floating point object representation.

Definition: <floatCharacterization>

Report the minimum denormalized value allowed by the characterized floating point object representation. This satisfies the ISO/IEC 10967 floating point characterization requirement $fmin_D$, and is equal to

```
self radix raisedTo: (self emin - self precision)
```

The result is unspecified if denormalized values are not allowed by the characterized representation.

Return Values

<Float> unspecified

Errors

none

5.6.8.9 Message: fminNormalized

Synopsis

The minimum normalized value allowed by the characterized floating point object representation.

Definition: <floatCharacterization>

Report the minimum normalized value allowed by the characterized floating point object representation. This satisfies the ISO/IEC 10967 floating point characterization requirement $fmin_N$, and is equal to

```
self radix raisedTo: (self emin - 1).
```

Return Values

<Float> unspecified

Errors

none

5.6.8.10 Message: pi

Synopsis

The closest floating point approximation to Pi.

Definition: <floatCharacterization>

Return the closest floating point approximation to Pi for the characterized floating point object representation.

Return Values

<Float> unspecified

Errors

none

5.6.8.11 Message: precision

Synopsis

The precision of the characterized floating point object representation.

Definition: <floatCharacterization>

Report the precision, the number of radix digits, of floating point objects of the characterized floating point object representation. This satisfies the ISO/IEC 10967 floating point characterization requirement *p*. The result must be greater than or equal to two.

Return Values

<integer>unspecified

Errors

none

5.6.8.12 Message: radix

Synopsis

The radix of the characterized floating point object representation.

Definition: <floatCharacterization>

Report the base, or radix, of the characterized floating point object representation. This satisfies the ISO/IEC 10967 floating point characterization requirement *r.* The result must be an even number greater than or equal to two.

Return Values

<integer>unspecified

Errors

none

5.6.9 Protocol: <Fraction factory>

Conforms To

<Object>

Description

Represents protocol for creating an exact representation for rational numbers.

Standard Globals

`Fraction`	Conforms to the protocol <Fraction factory>. Its language element type is implementation defined.

Messages

numerator:denominator:

5.6.9.1 Message: numerator: top denominator: bottom

Synopsis

Answer a new fraction whose numerator is `top`, and whose denominator is `bottom`.

Definition: <Fraction factory>

Answer a new fraction whose numerator is `top`, and whose denominator is `bottom`. It is unspecified whether the result is reduced to the smallest possible denominator. If (`top = bottom`) or (|`bottom` = 1) the result conforms to <integer> otherwise it conforms to <Fraction>. If `bottom` = 0 a `ZeroDivide` exception is signaled.

Parameters

`top`	<integer>	unspecified
`bottom`	<integer>	unspecified

Return Values

<Fraction>	unspecified
<integer>	unspecified

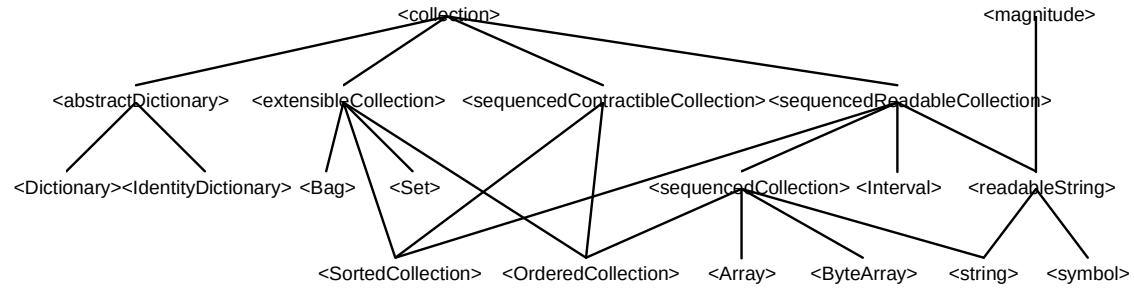
Errors

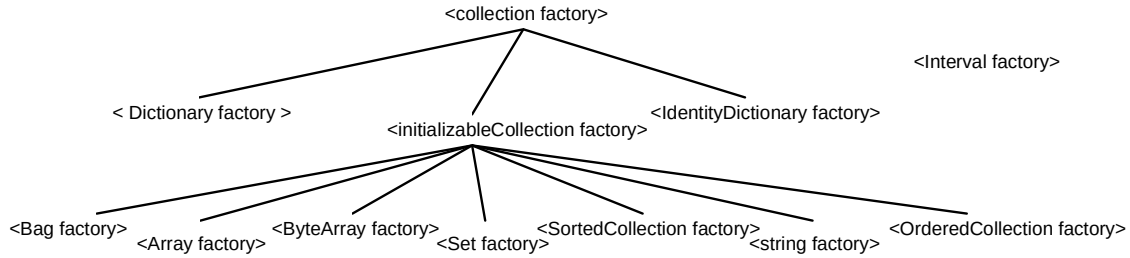
none

5.7 Collection Protocols

This section includes protocols that define the behavior of the standard collection classes.

The graphs below shows the conformance relationships between the protocols defined in this section (except for <magnitude>, which is contained in the section on numeric protocols).





5.7.1 Protocol: <collection>

Conforms To

<Object>

Description

Provides protocol for manipulating and operating on a collection of objects, called *elements*, either individually or as a whole. A collection can be fixed or variable sized, ordered or unordered, and its elements may or may not be accessible by external keys.

Some implementations of collections may choose to use the hash values, as defined by either the message `#hash` or the message `#identityHash`, of either the elements of the collection or the keys by which those elements are accessed (if there are any). If the hash values of such objects are modified, the behavior of any message sent to such a collection is undefined until the message `#rehash` has been sent to the collection in order to restore the consistency of the collection.

Rationale

#rehash message was moved to Collection to avoid any pre-existing implementation assumptions about its use in the implementation of collection. Any collection concievable might use hashing and hence could need to be rehashed.

Messages

allSatisfy:
anySatisfy:
asArray
asBag
asByteArray
asOrderedCollection
asSet
asSortedCollection
asSortedCollection:
collect:
detect:
detect:ifNone:
do:
do:separatedBy:
includes:
inject:into:
isEmpty
notEmpty
occurrencesOf:
rehash
reject:
select:
size

5.7.1.1 Message: allSatisfy: discriminator

Synopsis

Return *true* if the `discriminator` evaluates to *true* for every *element* of the receiver. Otherwise return *false*.

Definition: <collection>

Return *true* if the `discriminator` evaluates to *true* for every *element* of the receiver. Return *true* if the receiver is empty. Otherwise return *false*.

It is unspecified whether the `discriminator` will be evaluated with every *element* of the receiver.

Parameters

`discriminator` <monadicValuable> uncaptured

Return Values

<boolean> unspecified

Errors

If the *elements* of the receiver are inappropriate for use as arguments to `discriminator`.

If `discriminator` evaluates to an object that does not conform to the protocol <boolean> for each *element* of the receiver.

5.7.1.2 Message: anySatisfy: discriminator

Synopsis

Return *true* if the `discriminator` evaluates to *true* for any *element* of the receiver. Otherwise return *false*.

Definition: <collection>

Return *true* if the `discriminator` evaluates to *true* for any *element* of the receiver. Otherwise return *false*. Return *false* if the receiver is empty.

It is unspecified whether the `discriminator` will be evaluated with every *element* of the receiver.

Parameters

`discriminator` <monadicValuable> uncaptured

Return Values

<boolean> unspecified

Errors

If the *elements* of the receiver are inappropriate for use as arguments to `discriminator`.

If `discriminator` evaluates to an object that does not conform to the protocol <boolean> for any *element* of the receiver.

5.7.1.3 Message: asArray

Synopsis

Answer an array whose *elements* are the *elements* of the receiver.

Definition: <collection>

Answer an array with the same *elements* as the receiver. The result has the same size as the receiver, as defined by the `#size` message.

If the receiver maintains an ordering for its *elements*, the order of those *elements* will be preserved in the result.

Return Values

<Array> unspecified

Errors

none

5.7.1.4 Message: asBag

Synopsis

Answer a bag with the same *elements* as the receiver.

Definition: <collection>

Answer a bag with the same *elements* as the receiver.

The result is unspecified if the receiver contains *nil*.

Return Values

<Bag> unspecified

Errors

none

5.7.1.5 Message: asByteArray

Synopsis

Answer a byte array whose *elements* are the *elements* of the receiver.

Definition: <collection>

Answer a byte array with the same *elements* as the receiver. The result has the same size as the receiver, as defined by the `#size` message.

If the receiver maintains an ordering for its *elements*, the order of those *elements* will be preserved in the result.

Return Values

<ByteArray> unspecified

Errors

If any *elements* in the receiver are not integers with values between 0 and 255.

5.7.1.6 Message: asOrderedCollection

Synopsis

Answer an ordered collection whose *elements* are the *elements* of the receiver.

Definition: <collection>

Answer a ordered collection with the same *elements* as the receiver. The result has the same size as the receiver, as defined by the `#size` message.

If the receiver maintains an ordering for its *elements*, the order of those *elements* will be preserved in the result.

Return Values

<OrderedCollection> unspecified

Errors

none

5.7.1.7 Message: asSet

Synopsis

Answer a set with the same *elements* as the receiver.

Definition: <collection>

Answer a set with the same *elements* as the receiver. Since sets do not store duplicate elements, the result may have fewer elements than the receiver.

The result is undefined if the receiver contains *nil*.

Return Values

<Set> unspecified

Errors

none

5.7.1.8 Message: asSortedCollection

Synopsis

Answer a sorted collection with the same *elements* as the receiver.

Definition: <collection>

Answer a sorted collection with the same *elements* as the receiver. The default sort block is used.

Return Values

<SortedCollection> unspecified

Errors

If any *element* of the receiver is not appropriate as a parameter to the default sort block.

5.7.1.9 Message: asSortedCollection: sortBlock

Synopsis

Answer a sorted collection with the same *elements* as the receiver. The parameter `sortBlock` is used as the sort block.

Definition: <collection>

Answer a sorted collection with the same *elements* as the receiver. The parameter `sortBlock` is used as the sort block and must meet the requirements of a sort block as specified by <SortedCollection>.

Parameters

`sortBlock` <dyadicValuable> captured

Return Values

<SortedCollection> unspecified

Errors

If `sortBlock` does not meet the requirements for a sort block as specified by <SortedCollection>.

If any *element* of the receiver is not appropriate as a parameter to the `sortBlock`.

5.7.1.10 Message: collect: transformer

Synopsis

Answer a new collection constructed by gathering the results of evaluating `transformer` with each *element* of the receiver.

Definition: <collection>

For each *element* of the receiver, `transformer` is evaluated with the *element* as the parameter. The results of these evaluations are collected into a new collection.

The *elements* are traversed in the same order as they would be if the message `#do:` had been sent to the receiver.

Unless specifically refined, this message is defined to answer an object conforming to the same protocol as the receiver.

Parameters

`transformer` <monadicValuable> uncaptured

Return Values

<RECEIVER> new

Errors

If any *element* of the receiver is inappropriate for use as arguments to `transformer`.

If the result of evaluating the `transformer` does not conform to any *element type* restrictions of the collection to be returned.

5.7.1.11 Message: detect: discriminator

Synopsis

Return the first *element* of the receiver which causes `discriminator` to evaluate to *true* when the *element* is used as the argument.

Definition: <collection>

Return the first *element* of the receiver for which the `discriminator` evaluates to *true* when given that *element* as an argument. The `discriminator` will only be evaluated until such an object is found or until all of the *elements* of the collection have been used as arguments. That is, there may be *elements* of the receiver that are never used as arguments to the `discriminator`.

The *elements* are traversed in the same order as they would be if the message `#do:` had been sent to the receiver.

The result is undefined if `discriminator` does not evaluate to *true* for any *element*.

Parameters

`discriminator` <monadicValuable> uncaptured

Return Values

<Object> state

Errors

If the *elements* of the receiver are inappropriate for use as arguments to `discriminator`.

If `discriminator` evaluates to an object that does not conform to the protocol <boolean> for any *element* of the receiver.

5.7.1.12 Message: detect: discriminator ifNone: exceptionHandler

Synopsis

Return the first *element* of the receiver which causes `discriminator` to evaluate to *true* when used as the argument to the evaluation. Answer the result of evaluating `exceptionHandler` if no such *element* is found.

Definition: <collection>

Return the first *element* of the receiver for which the `discriminator` evaluates to *true* when given that *element* as an argument. The `discriminator` will only be evaluated until such an object is found or until all of the *elements* of the collection have been used as arguments. That is, there may be *elements* of the receiver that are never used as arguments to the `discriminator`.

The *elements* are traversed in the same order as they would be if the message `#do:` had been sent to the receiver.

If no element causes `discriminator` to evaluate to *true*, answer the result of `exceptionHandler value`.

Parameters

`discriminator`	<monadicValuable>	uncaptured
`exceptionHandler`	<niladicValuable>	uncaptured

Return Values

<Object> state

<Object> unspecified

Errors

If the *elements* of the receiver are inappropriate for use as arguments to `discriminator`.

If `discriminator` evaluates to an object that does not conform to the protocol <boolean> for any *element* of the receiver.

5.7.1.13 Message: do: operation

Synopsis

Evaluate `operation` with each *element* of the receiver.

Definition: <collection>

For each *element* of the receiver, `operation` is evaluated with the *element* as the parameter.

Unless specifically refined, the *elements* are not traversed in a particular order. Each *element* is visited exactly once. Conformant protocols may refine this message to specify a particular ordering.

Parameters

`operation`	<monadicValuable>	uncaptured

Return Values

UNSPECIFIED

Errors

If the *elements* of the receiver are inappropriate for use as arguments to `operation`.

5.7.1.14 Message: do: operation separatedBy: separator

Synopsis

Evaluate `operation` with each element of the receiver interspersed by evaluation of `separator`.

Definition: <collection>

For each element of the receiver, `operation` is evaluated with the element as the parameter. Before evaluating `operation` the second and subsequent times evaluate `separator`. `Separator` is not evaluated if there are less than two elements nor after the last element.

Parameters

`operation`	<monadicValuable>	uncaptured
`separator`	<niladicValuable>	uncaptured

Return Values

UNSPECIFIED

Errors

None

5.7.1.15 Message: includes: target

Synopsis

Answer *true* if an *element* of the receiver is *equivalent* to `target`. Answer *false* otherwise.

Definition: <collection>

This message is used to test an object for inclusion among the receiver's *elements*. Answer *true* if at least one of the receiver's *elements* is *equivalent* to `target`. Answer *false* otherwise.

Parameters

`target`	<Object>	uncaptured

Return Values

<boolean>	unspecified

Errors

none

5.7.1.16 Message: inject: initialValue into: operation

Synopsis

Answer the final result of evaluating `operation` using each *element* of the receiver and the previous evaluation result as the parameters.

Definition: <collection>

The first evaluation of `operation` is performed with `initialValue` as the first parameter, and the first *element* of the receiver as the second parameter. Subsequent evaluations are done with the result of the previous evaluation as the first parameter, and the next *element* as the second parameter. The result of the last evaluation is answered.

The *elements* are traversed in the same order as they would be if the message `#do:` had been sent to the receiver.

Parameters

`initialValue`	<Object>	uncaptured
`operation`	<dyadicValuable>	uncaptured

Return Values

<Object> unspecified

Errors

none

5.7.1.17 Message: isEmpty

Synopsis

Return *true* if the receiver contains no *elements*. Return *false* otherwise.

Definition: <collection>

Return *true* if and only if

```
receiver size = 0
```

is *true*. Otherwise return *false*.

Return Values

<boolean>	unspecified

Errors

none

5.7.1.18 Message: notEmpty

Synopsis

Return *true* if the receiver contains *elements*. Return *false* otherwise.

Definition: <collection>

Return true if the receiver contains elements. Return *false* otherwise. This is equivalent to

```
receiver isEmpty not
```

Return Values

<boolean>	unspecified

Errors

none

5.7.1.19 Message: occurrencesOf: target

Synopsis

Answer the number of *elements* of the receiver which are *equivalent* to `target`.

Definition: <collection>

Answer the number of *elements* of the receiver which are *equivalent* to `target`.

Parameters

`target` <Object> uncaptured

Return Values

<integer> unspecified

Errors

none

5.7.1.20 Message: rehash

Synopsis

Re-establish hash invariants, if any.

Definition: <collection>

Re-establish any hash invariants of the receiver.

Return Values

UNSPECIFIED

Errors

none

5.7.1.21 Message: reject: discriminator

Synopsis

Answer a new collection which includes only the *elements* in the receiver which cause `discriminator` to evaluate to *false*.

Definition: <collection>

For each *element* of the receiver, `discriminator` is evaluated with the *element* as the parameter. Each *element* which causes `discriminator` to evaluate to *false* is included in the new collection.

The *elements* are traversed in the same order as they would be if the message `#do:` had been sent to the receiver.

Unless specifically refined, this message is defined to answer an object conforming to the same protocol as the receiver. If both the receiver and the result maintain an ordering of their *elements*, the *elements* of the result will be in the same relative order as the *elements* of the receiver.

Parameters

`discriminator` <monadicValuable> uncaptured

Return Values

<RECEIVER> new

Errors

If the *elements* of the receiver are inappropriate for use as arguments to `discriminator`.

If `discriminator` evaluates to an object that does not conform to the protocol <boolean> for any *element* of the receiver.

5.7.1.22 Message: select: discriminator

Synopsis

Answer a new collection which contains only the *elements* in the receiver which cause `discriminator` to evaluate to *true*.

Definition: <collection>

For each *element* of the receiver, `discriminator` is evaluated with the *element* as the parameter. Each *element* which causes `discriminator` to evaluate to *true* is included in the new collection.

The *elements* are traversed in the same order as they would be if the message `#do:` had been sent to the receiver.

Unless specifically refined, this message is defined to answer an object conforming to the same protocol as the receiver. If both the receiver and the result maintain an ordering of their *elements*, the *elements* of the result will be in the same relative order as the *elements* of the receiver.

Parameters

`discriminator` <monadicValuable> uncaptured

Return Values

<RECEIVER> new

Errors

If the elements of the receiver are inappropriate for use as arguments to discriminator.

If `discriminator` evaluates to an object that does not conform to the protocol <boolean> for any *element* of the receiver.

5.7.1.23 Message: size

Synopsis

Answer the number of *elements* in the receiver.

Definition: <collection>

Answer the number of *elements* in the receiver.

Return Values

<integer> unspecified

Errors

none

5.7.2 Protocol: <abstractDictionary>

Conforms To

<collection>

Description

Provides protocol for accessing, adding, removing, and iterating over the *elements* of an unordered collection whose *elements* are accessed using an explicitly assigned external *key*.

Glossary Entries

Messages

addAll:
at:
at:ifAbsent:
at:ifAbsentPut:
at:put:
collect:
includesKey:
keyAtValue:
keyAtValue:ifAbsent:
keys
keysAndValuesDo:
keysDo:
reject:
removeAllKeys:
removeAllKeys:ifAbsent:
removeKey:
removeKey:ifAbsent:
select:
values

5.7.2.1 Message: addAll: dictionary

Synopsis

Store the *elements* of `dictionary` in the receiver at the corresponding *keys* from `dictionary`.

Definition: <abstractDictionary>

This message is equivalent to repeatedly sending the `#at:put:` message to the receiver with each of the *keys* and elements in `dictionary` in turn. If a *key* in `dictionary` is *key equivalent* to a *key* in the receiver, the associated element in `dictionary` replaces the element in the receiver.

Parameters

`dictionary` <abstractDictionary> unspecified

Return Values

UNSPECIFIED

Errors

none

5.7.2.2 Message: at: key

Synopsis

Answer the *element* at *`key`* in the receiver.

Definition: <abstractDictionary>

This message defines *element* lookup based on a `key`. Answer the *element* stored at `key`.

Lookup is successful if an *element* has been previously stored in the receiver at a *key* that is *key equivalent* to `key`. This element is answered. Specifically, the following expression must return *true* for all appropriate bindings of `dictionary`, `key`, and `value`:

```
dictionary at: key put: value.
^(dictionary at: key) == value
```

The result is undefined if the receiver does not contain an *element* keyed by `key` or if the `key` is *nil*.

Parameters

`key` <Object> uncaptured

Return Values

<Object> state

Errors

none

5.7.2.3 Message: at: key ifAbsent: operation

Synopsis

Answer the *element* at `key` in the receiver. If key lookup for `key` fails, then answer the result of evaluating `operation`.

Definition: <abstractDictionary>

Answer the *element* stored at the specified `key` if *key lookup* is successful. If the *key lookup* fails, answer the result of evaluating `operation` with no parameters.

The result is undefined if the `key` is *nil*.

Parameters

`key`	<Object>	uncaptured
`operation`	<niladicValuable>	uncaptured

Return Values

<Object> state

<ANY> unspecified

Errors

none

5.7.2.4 Message: at: key ifAbsentPut: operation

Synopsis

Answer the *element* at `key` in the receiver. If *key lookup* for `key` fails, then store and return the result of evaluating `operation`.

Definition: <abstractDictionary>

This message is the same as the `#at:` message if *key lookup* is successful. If the *key lookup* fails, the result of evaluating `operation` with no parameters is added at `key` and answered.

The result is undefined if the `key` is *nil*.

Parameters

`key`	<Object>	unspecified
`operation`	<niladicValuable>	uncaptured

Return Values

<Object> state

Errors

none

5.7.2.5 Message: at: key put: newElement

Synopsis

Store `newElement` at `key` in the receiver. Answer `newElement`.

Definition: <abstractDictionary>

If lookup succeeds for `key`, then `newElement` replaces the *element* previously stored at `key`. Otherwise, the `newElement` is stored at the new `key`. In either case, subsequent successful lookups for `key` will answer `newElement`. Answer `newElement`.

The result is undefined if the `key` is *nil*.

Parameters

`key`	<Object>	unspecified
`newElement`	<Object>	captured

Return Values

<Object> state

Errors

none

5.7.2.6 Message Refinement: collect: transformer

Synopsis

Answer a new collection constructed by gathering the results of evaluating `transformer` with each *element* of the receiver.

Definition: <collection>

For each *element* of the receiver, `transformer` is evaluated with the *element* as the parameter. The results of these evaluations are collected into a new collection.

The *elements* are traversed in the order specified by the `#do:` message for the receiver.

Unless specifically refined, this message is defined to answer an object conforming to the same protocol as the receiver.

Refinement: <abstractDictionary>

Answer a new instance of the receiver's type with the same *keys*. For each *key* of the answer, a new element is obtained by evaluating `transformer` with the corresponding element of the receiver as the parameter.

Parameters

`transformer`	<monadicValuable>	uncaptured

Return Values

<RECEIVER>	new

Errors

If the *elements* of the receiver are inappropriate for use as arguments to `transformer`.

If the result of evaluating the `transformer` does not conform to any *element type* restrictions of the collection to be returned.

5.7.2.7 Message: includesKey: key

Synopsis

Answer *true* if the receiver contains an *element* stored at `key`. Answer *false* otherwise.

Definition: <abstractDictionary>

Answer *true* if the *key lookup* for the `key` succeeds. Answer *false* otherwise.

The result is undefined if the `key` is *nil*.

Parameters

`key`	<Object>	uncaptured

Return Values

<boolean>	unspecified

Errors

none

5.7.2.8 Message: keyAtValue: value

Synopsis

Answer a *key* such that the *element* stored at this *key* is equal to `value`. Answer *nil* if no such *key* is found.

Definition: <abstractDictionary>

Answer an object such that *key lookup* with this object will answer an *element* in the receiver *equivalent* to `value`. Note that if there are multiple *elements* in the receiver that are *equivalent* to `value`, then the one whose *key* is answered is arbitrary.

The result is undefined if the receiver does not contain an *element equivalent* to `value`.

Parameters

`value` <Object> uncaptured

Return Values

<Object> state

Errors

none

5.7.2.9 Message: keyAtValue: value ifAbsent: operation

Synopsis

Answer a *key* such that the element stored at this *key* is *equivalent* to `value`. Answer the result of evaluating `operation` if no such *key* is found.

Definition: <abstractDictionary>

Answer an object such that *key lookup* with this object will answer an *element* in the receiver *equivalent* to `value`. If no *element equivalent* to `value` is found, then the result of evaluating `operation` with no parameters is answered.

Parameters

`value` <Object> uncaptured

`operation` <niladicValuable> uncaptured

Return Values

<Object> state

<ANY> unspecified

Errors

none

5.7.2.10 Message: keys

Synopsis

Answer a collection of *keys* at which there is an *element* stored in the receiver.

Definition: <abstractDictionary>

Answer a collection of all the *keys* in the receiver. The size of the result is equal to the size of the receiver.

Return Values

<collection> unspecified

Errors

none

5.7.2.11 Message: keysAndValuesDo: operation

Synopsis

Iteratively evaluate `operation` with each of the receiver's *keys* and values.

Definition: <abstractDictionary>

For each *element* in the receiver, `operation` is evaluated with the corresponding *key* as the first argument and the *element* as the second argument.

The order in which the *elements* are traversed is not specified. Each *key* is visited exactly once.

Parameters

`operation`	<dyadicValuable>	uncaptured

Return Values

UNSPECIFIED

Errors

If any of the *keys* or values are not appropriate as an argument to operation.

5.7.2.12 Message Refinement: keysDo: operation

Synopsis

Iteratively evaluate `operation` with each of the receiver's *keys* at which there are *elements* stored.

Definition: <abstractDictionary>

For each *key* in the receiver, `operation` is evaluated with the *key* used as the parameter.

The order in which the *elements* are traversed is not specified. Each *key* is visited exactly once.

Parameters

`operation`	<monadicValuable>	uncaptured

Return Values

UNSPECIFIED

Errors

If any of the *keys* are not appropriate as an argument to operation.

5.7.2.13 Message Refinement: reject: discriminator

Synopsis

Answer a new collection which excludes the *elements* in the receiver which cause `discriminator` to evaluate to *true*.

Definition: <collection>

For each *element* of the receiver, `discriminator` is evaluated with the *element* as the parameter. Each *element* which causes `discriminator` to evaluate to *false* is added to the new collection.

The *elements* are traversed in the order specified by the `#do:` message for the receiver.

Unless specifically refined, this message is defined to answer an object conforming to the same protocol as the receiver.

Refinement: <abstractDictionary>

For each *key* of the receiver, `discriminator` is evaluated with the corresponding *element* as the parameter. If the *element* causes `discriminator` to evaluate to *false*, the *key* is added to the answer with the *element* as its corresponding value.

Parameters

`discriminator` <monadicValuable> uncaptured

Return Values

<RECEIVER> new

Errors

If the *elements* of the receiver are inappropriate for use as arguments to `discriminator`.

If `discriminator` evaluates to an object that does not conform to the protocol <boolean> for any *element* of the receiver.

5.7.2.14 Message: removeAllKeys: keys

Synopsis

Remove any *elements* from the receiver which are stored at the *keys* specified in `keys`.

Definition: <abstractDictionary>

This message has the same effect on the receiver as repeatedly sending the `#removeKey:` message for each *element* in `keys`.

The result is undefined if duplicate *keys,* as defined by *key equivalence*, are in the `keys` or if any *element* in `keys` is not a valid *key* of the receiver.

Parameters

`keys` <collection> uncaptured

Return Values

UNSPECIFIED

Errors

none

5.7.2.15 Message: removeAllKeys: keys ifAbsent: operation

Synopsis

Remove any *elements* from the receiver which are stored at the *keys* specified in `keys`. For any *element* in `keys` which is not a valid *key* of the receiver, evaluate `operation` with that *element* as the argument, but do not stop the enumeration.

Definition: <abstractDictionary>

This message has the same effect on the receiver as repeatedly sending the `#removeKey:ifAbsent:` message for each *element* in `keys`. If any *element* in `keys` is not a valid *key* of the receiver, evaluate `operation` with that *element* as the parameter and continue the enumeration.

Parameters

`keys` <collection> uncaptured

`operation` <monadicValuable> uncaptured

Return Values

UNSPECIFIED

Errors

If any *element* of `keys` is not a valid *key* of the receiver and inappropriate for use as an argument to the `operation`.

5.7.2.16 Message: removeKey: key

Synopsis

Remove the *element* which is stored at `key` in the receiver. Answer the removed element.

Definition: <abstractDictionary>

This message defines removal of a *key* from the receiver. If *key lookup* for `key` is successful, then both `key` and its corresponding *element* are removed. Answer the removed *element*.

The result is undefined if the receiver does not contain an *element* keyed by `key`.

The result is undefined if the `key` is *nil*.

Parameters

`key`	<Object>	uncaptured

Return Values

<Object> state

Errors

none

5.7.2.17 Message: removeKey: key ifAbsent: operation

Synopsis

Remove the *element* which is stored at `key` in the receiver and answer the removed *element*. Answer the result of evaluating `operation` if no such *key* is found in the receiver.

Definition: <abstractDictionary>

If *key lookup* for `key` is successful, then both `key` and its corresponding *element* are removed. Answer the removed *element*.

If the *key lookup* fails, the result of evaluating `operation` with no parameters is answered.

The result is undefined if the `key` is *nil*.

Parameters

`key`	<Object>	uncaptured
`operation`	<niladicValuable>	uncaptured

Return Values

<Object> state

<ANY> unspecified

Errors

none

5.7.2.18 Message Refinement: select: discriminator

Synopsis

Answer a new collection which contains the *elements* in the receiver which cause `discriminator` to evaluate to *true*.

Definition: <collection>

For each *element* of the receiver, `discriminator` is evaluated with the *element* as the parameter. Each *element* which causes `discriminator` to evaluate to *true* is added to the new collection.

The *elements* are traversed in the order specified by the `#do:` message for the receiver.

Unless specifically refined, this message is defined to answer an object conforming to the same protocol as the receiver.

Refinement: <abstractDictionary>

For each *key* of the receiver, `discriminator` is evaluated with the *element* as the parameter. If *element* causes `discriminator` to evaluate to *true*, the *key* is added to the answer with value element.

If `discriminator` evaluates to an object that does not conform to the protocol <boolean> for any *element* of the receiver.

Parameters

`discriminator` <monadicValuable> uncaptured

Return Values

<RECEIVER> new

Errors

If the elements of the receiver are inappropriate for use as arguments to `discriminator`.

5.7.2.19 Message: values

Synopsis

Answer a collection of the receiver's *elements.*

Definition: <abstractDictionary>

Answer a collection of the receiver's *elements.*

Return Values

<sequencedReadableCollection> unspecified

Errors

none

5.7.3 Protocol: <Dictionary>

Conforms To

<abstractDictionary>

Description

Represents an unordered collection whose *elements* can be accessed using an explicitly assigned external *key*. *Key equivalence* is defined as sending the #= message.

Messages

none

5.7.4 Protocol: <IdentityDictionary>

Conforms To

<abstractDictionary>

Description

This protocol defines the behavior of unordered collections whose *elements* can be accessed using an explicitly-assigned, external *key*. *Key equivalence* is defined as sending the #== message.

Messages

none

5.7.5 Protocol: <extensibleCollection>

Conforms To

<collection>

Description

Provides protocol for adding *elements* to and removing *elements* from a variable sized collection.

Messages

add:
addAll:
remove:
remove:ifAbsent:
removeAll:

5.7.5.1 Message: add: newElement

Synopsis

Add `newElement` to the receiver's *elements*.

Definition: <extensibleCollection>

This message adds a `newElement` to the receiver. Unless specifically refined, the position of the `newElement` in the *element* traversal order is unspecified.

Conformant protocols may place restrictions on the type of objects that are valid *elements*. Unless otherwise specified, any object is acceptable.

Parameters

`newElement` <Object> captured

Return Values

UNSPECIFIED

Errors

none

5.7.5.2 Message: addAll: newElements

Synopsis

Add each *element* of `newElements` to the receiver's *elements*.

Definition: <extensibleCollection>

This message adds each *element* of `newElements` to the receiver.

The operation is equivalent to adding each *element* of `newElements` to the receiver using the `#add:` message with the *element* as the parameter. The `newElements` are traversed in the order specified by the `#do:` message for `newElements`.

Parameters

`newElements`	<collection>	unspecified

Return Values

UNSPECIFIED

Errors

none

5.7.5.3 Message: remove: oldElement

Synopsis

Remove the first *element* of the receiver which is *equivalent* to `oldElement` and return the removed *element*.

Definition: <extensibleCollection>

Remove the first *element* of the receiver which is *equivalent* to `oldElement` and return the removed *element*.

The *elements* are tested in the same order in which they would be enumerated by the message `#do:` for this receiver.

The behavior is undefined if an object *equivalent* to `oldElement` is not found.

Parameters

`oldElement`	<Object>	uncaptured

Return Values

<Object> state

Errors

none

5.7.5.4 Message: remove: oldElement ifAbsent: exceptionHandler

Synopsis

Remove the first *element* of the receiver which is *equivalent* to `oldElement`. If it is not found, answer the result of evaluating `exceptionHandler`.

Definition: <extensibleCollection>

The first *element* of the receiver which is *equivalent* to `oldElement` is removed from the receiver's *elements*. If no such *element* is found, answer the result of evaluating `exceptionHandler` with no parameters.

The *elements* are tested in the same order in which they would be enumerated by the message `#do:` for this receiver.

Parameters

`oldElement`	<Object>	uncaptured
`exceptionHandler`	<niladicValuable>	uncaptured

Return Values

<Object> state

<Object> unspecified

Errors

none

5.7.5.5 Message: removeAll: oldElements

Synopsis

For each *element* in `oldElements`, remove the first *element* from the receiver which is *equivalent* to this *element*.

Definition: <extensibleCollection>

This message is used to remove each *element* of a given collection from the receiver's *elements*. The operation is defined to be equivalent to removing each *element* of `oldElements` from the receiver using the `#remove:` message with the *element* as the parameter.

The behavior is undefined if any *element* of `oldElements` is not found.

Parameters

`oldElements` <collection> uncaptured

Return Values

UNSPECIFIED

Errors:

none

5.7.6 Protocol: <Bag>

Conforms To

<extensibleCollection>

Description

Represents an unordered, variable sized collection whose *elements* can be added or removed, but cannot be individually accessed by external *keys*. A bag is similar to a set but can contain duplicate *elements*. *Elements* are duplicates if they are *equivalent*.

Messages

add:
add:withOccurrences:
addAll:
collect:

5.7.6.1 Message Refinement: add: newElement

Synopsis

Add `newElement` to the receiver's *elements*.

Definition: <extensibleCollection>

This message adds a `newElement` to the receiver. Unless specifically refined, the position of the `newElement` in the *element* traversal order is unspecified.

Conformant protocols may place restrictions on the type of objects that are valid *elements*. Unless otherwise specified, any object is acceptable.

Refinement: <Bag>

The result is undefined if `newElement` is *nil*.

Parameters

`newElement`	<Object>	captured

Return Values

UNSPECIFIED

Errors

none

5.7.6.2 Message: add: newElement withOccurrences: count

Synopsis

Add `newElement` `count` times to the receiver's *elements*.

Definition: <Bag>

This message adds an *element* to the receiver multiple times. The operation is equivalent to adding `newElement` to the receiver `count` times using the `#add:` message with `newElement` as the parameter.

The result is undefined if `newElement` is *nil*.

Parameters

`newElement`	<Object>	captured
`count`	<integer>	unspecified

Return Values

UNSPECIFIED

Errors

none

5.7.6.3 Message Refinement: addAll: newElements

Synopsis

Add each element of `newElements` to the receiver's *elements*.

Definition: <extensibleCollection>

This message adds each *element* of `newElements` to the receiver.

The operation is equivalent to adding each *element* of `newElements` to the receiver using the `#add:` message with the *element* as the parameter. The `newElements` are traversed in the order specified by the `#do:` message for `newElements`.

Refinement: <Bag>

The result is undefined if `newElements` contains *nil*.

The traversal order is unspecified.

Parameters

`newElements`	<collection>	uncaptured

Return Values

UNSPECIFIED

Errors

none

5.7.6.4 Message Refinement: collect: transformer

Synopsis

Answer a new collection constructed by gathering the results of evaluating `transformer` with each *element* of the receiver.

Definition: <collection>

For each *element* of the receiver, `transformer` is evaluated with the *element* as the parameter. The results of these evaluations are collected into a new collection.

The *elements* are traversed in the order specified by the `#do:` message for the receiver.

Unless specifically refined, this message is defined to answer an objects conforming to the same protocol as the receiver.

Refinement: <Bag>

The result is undefined if `transformer` evaluates to *nil* for any *element* of the receiver.

Parameters

`transformer` <monadicValuable> uncaptured

Return Values

<RECEIVER> new

Errors

If the *elements* of the receiver are inappropriate for use as arguments to `transformer`.

If the result of evaluating the `transformer` does not conform to any *element type* restrictions of the collection to be returned.

5.7.7 Protocol: <Set>

Conforms To

<extensibleCollection>

Description

Represents an unordered, variable sized collection whose *elements* can be added or removed, but cannot be individually accessed by external *keys*. A set is similar to a bag but cannot contain duplicate *elements*.

Messages

add:
addAll:
collect:

5.7.7.1 Message Refinement: add: newElement

Synopsis

Add `newElement` to the receiver's *elements*.

Definition: <extensibleCollection>

This message adds a `newElement` to the receiver. Unless specifically refined, the position of the `newElement` in the *element* traversal order is unspecified.

Conformant protocols may place restrictions on the type of objects that are valid *elements*. Unless otherwise specified, any object is acceptable.

Refinement: <Set>

Since sets may not contain duplicates, if there is already an *element* in the receiver that is *equivalent* to `newElement`, this operation has no effect.

The results are undefined if `newElement` is *nil*.

The equivalence of `newElement` with respect to other objects should not be changed while `newElement` is in the collection, as this would violate the invariant under which the element was placed within the collection.

Parameters

`newElement` <Object> captured

Return Values

UNSPECIFIED

Errors

none

5.7.7.2 Message Refinement: addAll: newElements

Synopsis

Add each *element* of `newElements` to the receiver's *elements*.

Definition: <extensibleCollection>

This message adds each *element* of `newElements` to the receiver.

The operation is equivalent to adding each *element* of `newElements` to the receiver using the `#add:` message with the *element* as the parameter. The `newElements` are traversed in the order specified by the `#do:` message for `newElements`.

Refinement: <Set>

Duplicates will not be added.

The results are undefined if `newElements` contains *nil*.

Parameters

`newElements` <collection> unspecified

Return Values

UNSPECIFIED

Errors

none

5.7.7.3 Message Refinement: collect: transformer

Synopsis

Answer a new collection constructed by gathering the results of evaluating `transformer` with each *element* of the receiver.

Definition: <collection>

For each *element* of the receiver, `transformer` is evaluated with the *element* as the parameter. The results of these evaluations are collected into a new collection.

The *elements* are traversed in the order specified by the `#do:` message for the receiver.

Unless specifically refined, this message is defined to answer an objects conforming to the same protocol as the receiver.

Refinement: <Set>

Duplicates will not be added.

The results are undefined if `newElements` contains *nil*.

Parameters

`transformer` <monadicValuable> uncaptured

Return Values

<RECEIVER> new

Errors

If the *elements* of the receiver are inappropriate for use as arguments to `transformer`.

If the result of evaluating the `transformer` does not conform to any *element type* restrictions of the collection to be returned.

5.7.8 Protocol: <sequencedReadableCollection>

Conforms To

<collection>

Description

Provides protocol for reading an ordered collection of objects whose *elements* can be accessed using external integer *keys*. The *keys* are between one (1) and the number of *elements* in the collection, inclusive.

Messages

,
=
after:
at:
at:ifAbsent:
before:
copyFrom:to:
copyReplaceAll:with:
copyReplaceFrom:to:with:
copyReplaceFrom:to:withObject:
copyReplacing:withObject:
copyWith:
copyWithout:
do:
findFirst:
findLast:
first
from:to:do:
from:to:keysAndValuesDo:
indexOf:
indexOf:ifAbsent:
indexOfSubCollection:startingAt:
indexOfSubCollection:startingAt:ifAbsent:
keysAndValuesDo:
last
reverse
reverseDo:
with:do:

5.7.8.1 Message: , operand

Synopsis

Answer a new collection which is the concatenation of the receiver and `operand`.

Definition: <sequenceReadableCollection>

Answer a new collection containing all of the receiver's *elements* in their original order followed by all of the *elements* of `operand`, in their original order. The size of the new collection is equal to the sum of the sizes of the receiver and `operand`, as defined by the `#size` message.

Collections that enforce an ordering on their *elements* are permitted to refine this message to reorder the result.

Parameters

`operand` <sequencedReadableCollection> uncaptured

Return Values

<RECEIVER> new

Errors

If the *elements* of `operand` are not suitable for storage in instances of the receiver's class.

5.7.8.2 Message Refinement: = comparand

Synopsis

Object equivalence test.

Definition: <Object>

This message tests whether the receiver and the `comparand` are equivalent objects at the time the message is processed. Answer *true* if the receiver is equivalent to `comparand`. Otherwise answer *false*.

The meaning of "equivalent" cannot be precisely defined but the intent is that two objects are considered equivalent if they can be used interchangeably. Conformant protocols may choose to more precisely define the meaning of "equivalent".

The value of

```
receiver = comparand
```

is *true* if and only if the value of

```
comparand = receiver
```

would also be *true*. If the value of

```
receiver = comparand
```

is *true* then the receiver and `comparand` must have *equivalent hash value*s. Or more formally:

```
receiver = comparand
receiver hash = comparand hash
```

The equivalence of objects need not be *temporally invariant*. Two independent invocations of #= with the same receiver and `operand` objects may not always yield the same results. However, only objects whose implementation of #= is *temporally invariant* can be reliably stored within collections that use #= to discriminate objects.

Refinement: <sequenceReadableCollection>

Unless specifically refined, the receiver and `operand` are *equivalent* if all of the following are true:

1. The receiver and `operand` are instances of the same class.
2. They answer the same value for the `#size` message.

3. For all indices of the receiver, the *element* in the receiver at a given index is *equivalent* to the *element* in `operand` at the same index.

Element lookup is defined by the `#at:` message for the receiver and `operand`.

Parameters

`comparand` <Object> uncaptured

Return Values

<boolean> unspecified

Errors

none

5.7.8.3 Message: after: target

Synopsis

Answer the object immediately following the first *element* which is *equivalent* to `target` in the receiver.

Definition: <sequenceReadableCollection>

Answer the object immediately following the first *element* which is *equivalent* to `target` in the receiver. An *element* immediately follows another if its index is one greater than that of the other. The order used to determine which of the receiver's *elements* is the first to be *equivalent* to `target` is the traversal order defined by `#do:` for the receiver.

It is an error if the first occurrence of `target` is the last *element* of the receiver, or if the receiver does not include `target`.

Parameters

`target` <Object> uncaptured

Return Values

<Object> state

Errors

If there is no *element* in the receiver which is *equivalent* to `target`.

If the *element* which is equal to `target` is the last *element* in the receiver.

5.7.8.4 Message: at: index

Synopsis

Answer the *element* at the position `index` in the receiver.

Definition: <sequenceReadableCollection>

This message defines *element* retrieval based on an index. Answer the *element* at the specified `index`. The result is undefined if the receiver has no *element* at position `index`.

Parameters

`index` <integer> uncaptured

Return Values

<Object> state

Errors

If `index` is <= 0.

If `index` is greater than the receiver's size.

5.7.8.5 Message: at: index ifAbsent: exceptionBlock

Synopsis

Answer the *element* at the position `index` in the receiver. If there is no position corresponding to `index` in the receiver, then answer the result of evaluating `exceptionBlock`.

Definition: <sequenceReadableCollection>

This message defines *element* retrieval based on an index. Answer the *element* at the specified `index`. If there is no position corresponding to `index` in the receiver, then answer the result of evaluating `exceptionBlock`.

Parameters

`index`	<integer>	uncaptured
`exceptionBlock`	<niladicValuable>	uncaptured

Return Values

<Object> state

<ANY> unspecified

Errors

none

5.7.8.6 Message: before: target

Synopsis

Answer the object immediately preceding the first *element* which is *equivalent* to `target` in the receiver.

Definition: <sequenceReadableCollection>

Answer the object immediately preceding the first *element* which is *equivalent* to `target` in the receiver. An *element* immediately precedes another if its index is one less than that of the other.

It is an error if `target` is the first *element* of the receiver, or if the receiver does not include `target`.

Parameters

`target`	<Object>	uncaptured

Return Values

<Object> state

Errors

If there is no *element* in the receiver which is *equivalent* to `target`.

If the *element* which is equal to `target` is the first *element* in the receiver.

5.7.8.7 Message: copyFrom: start to: stop

Synopsis

Answer a new collection containing all of the *elements* of the receiver between the indices `start` and `stop` inclusive. If `stop` < `start`, the result has a size of zero.

Definition: <sequenceReadableCollection>

Answer a new collection containing the specified range of *elements* of the receiver in their original order. The *element* at index `start` in the receiver is at index 1 in the new collection; the *element* at index `start`+1 is at index 2, etc. If `stop` is less than `start`, then the new collection is empty. Otherwise, the size of the new collection is the maximum of (`stop` - `start` + 1) and 0.

The parameters `start` and `stop` must be positive..

Parameters

`start`	<integer>	uncaptured
`stop`	<integer>	uncaptured

Return Values

<RECEIVER> new

Errors

If `stop` >= `start` and (`start` < 1 or `start` > the receiver's size).

If `stop` >= `start` and (`stop` < 1 or `stop` > the receiver's size).

5.7.8.8 Message: copyReplaceAll: targetElements with: replacementElements

Synopsis

Answer a new collection in which all subsequences of *elements* in the receiver matching `targetElements` are replaced in the new collection by the *elements* in `replacementElements`.

Definition: <sequenceReadableCollection>

Answer a new collection with the *elements* of the receiver in their original order, except where a subsequence in the receiver matches `targetElements`. A subsequence in the receiver is said to match the *elements* of `targetElements` if:

1. They have the same number of *elements*.

2. For all indices of the subsequence, the *element* in the subsequence at a given index is *equivalent* to the *element* in `targetElements` at the same index.

Starting with the first *element* of the receiver and proceeding through ascending *elements*, each non-overlapping subsequence of the receiver matching `targetElements` is detected. The result is a copy of the receiver with each detected subsequence replaced by the sequence of *elements* of `replacementElements`.

Collections that enforce an ordering on their *elements* are permitted to refine this message to reorder the result.

Parameters

`targetElements`	<sequencedReadableCollection>	uncaptured
`replacementElements`	<sequencedReadableCollection>	uncaptured

Return Values

<RECEIVER> new

Errors

If any of the *elements* in `replacementElements` does not conform to any *element type* restrictions of instances of the receiver's class.

5.7.8.9 Message: copyReplaceFrom: start to: stop with: replacementElements

Synopsis

Answer a new collection, containing the same *elements* as the receiver, but with the *elements* in the receiver between `start` and `stop` inclusive replaced by the *elements* in `replacementElements`.

Definition: <sequenceReadableCollection>

This message can be used to insert, append, or replace. The size of `replacementElements` (as defined by `#size`) need not be the same as the number of *elements* being replaced. There are three cases:

1. If `stop` = `start` - 1 , and `start` is less than or equal to the size of the receiver, then the `replacementElements` are inserted between the *elements* at index `stop` and `start`. None of the receiver's *elements* are replaced.

2. If `stop` = the size of the receiver and `start` = `stop` + 1, then the operation is an append, and the `replacementElements` are placed at the end of the new collection.

3. Otherwise, the operation is a replacement, and the receiver's *elements* in the given range are replaced by the *elements* from `replacementElements`.

In all cases, the resulting collection consists of the receiver's *elements* from indices 1 to `start` - 1 in their original order, followed by the *elements* of `replacementElements`, followed by the remainder of the receiver's *elements* from index `stop` + 1 in their original order. The size of the result is the receiver's size - (`stop` - `start` + 1) + the `replacementElements` size.

The parameters `start` and `stop` must be positive.

Collections that enforce an ordering on their *elements* are permitted to refine this message to reorder the result.

Parameters

`start`	<integer>	uncaptured
`stop`	<integer>	uncaptured
`replacementElements`	<sequencedReadableCollection>	uncaptured

Return Values

<RECEIVER> new

Errors

The *elements* in `replacementElements` are not suitable for storage in instances of the receiver's class.

`start` > receiver's size + 1

`start` < 1

`stop` > receiver's size

`stop` < `start` - 1

5.7.8.10 Message: copyReplaceFrom: start to: stop withObject: replacementElement

Synopsis

Answer a new collection conforming to the same protocols as the receiver, in which the *elements* of the receiver between `start` and `stop` inclusive have been replaced with `replacementElement`.

Definition: <sequenceReadableCollection>

This message can be used to insert, append, or replace. There are three cases:

1. If `stop` = `start` - 1 , and `start` is less than or equal to the size of the receiver, then `replacementElement` is inserted between the *elements* at index `stop` and `start`. None of the receiver's *elements* are replaced.

2. If `stop` = the size of the receiver and `start` = `stop` + 1, then the operation is an append, and `replacementElement` is placed at the end of the new collection.

3. Otherwise, the operation is a replacement, and each of the receiver's *elements* in the given range is replaced by `replacementElement`.

The parameters `start` and `stop` must be non-negative.

Collections that by definition enforce an ordering on their *elements* are permitted to refine this message to reorder the result.

Parameters

`start`	<integer>	uncaptured
`stop`	<integer>	uncaptured
`replacementElement`	<Object>	uncaptured

Return Values

<RECEIVER> new

Errors

The `replacementElement` is not suitable for storage in instances of the receiver's class.

`start` > receiver's size + 1

`start` < 1

`stop` > receiver's size

`stop` < `start` - 1

5.7.8.11 Message: copyReplacing: targetElement withObject: replacementElement

Synopsis

Answer a new collection conforming to the same protocols as the receiver, in which any occurrences of `targetElement` are replaced by `replacementElement`.

Definition: <sequenceReadableCollection>

A new collection is created and initialized with the same *elements* as the receiver in the same order, except that any objects in the receiver which are *equivalent* to `targetElement` are replaced in the new collection by `replacementElement`.

Collections that enforce an ordering on their *elements* are permitted to refine this message to reorder the result.

Parameters

`targetElement`	<Object>	uncaptured
`replacementElement`	<Object>	uncaptured

Return Values

<RECEIVER>	new

Errors

If the `replacementElement` is inappropriate for storage in instances of the receiver's class.

5.7.8.12 Message: copyWith: newElement

Synopsis

Answer a new collection containing the same *elements* as the receiver, with `newElement` added.

Definition: <sequenceReadableCollection>

Answer a new collection with size one greater than the size of the receiver containing the *elements* of the receiver and `newElement` placed at the end.

Unless specifically refined, this message is defined to answer an instance of the same class as the receiver.

Collections that enforce an ordering on their *elements* are permitted to refine this message to reorder the result.

Parameters

`newElement`	<Object>	captured

Return Values

<RECEIVER>	new

Errors

none

5.7.8.13 Message Refinement: copyWithout: oldElement

Synopsis

Answer a new collection, containing the same *elements* as the receiver in their original order omitting any *elements equivalent* to `oldElement`.

Definition: <sequenceReadableCollection>

Answer a new collection with all of the *elements* of the receiver that are not *equivalent* to `oldElement`, in their original order.

Parameters

`oldElement`	<Object>	uncaptured

Return Values

<RECEIVER>	new

Errors

none

5.7.8.14 Message Refinement: do: operation

Synopsis

Evaluate `operation` with each *element* of the receiver.

Definition: <collection>

For each *element* of the receiver, `operation` is evaluated with the *element* as the parameter.

Unless specifically refined, the *elements* are not traversed in a particular order. Each *element* is visited exactly once. Conformant protocols may refine this message to specify a particular ordering.

Refinement: <sequenceReadableCollection>

The `operation` is evaluated with each *element* of the receiver in indexed order starting at 1. The first *element* is at index 1, the second at index 2, etc. The index of the last *element* is equal to the receiver's size.

Parameters

`operation`	<monadicValuable>	uncaptured

Return Values

UNSPECIFIED

Errors

If the *elements* of the receiver are inappropriate for use as arguments to `operation`.

5.7.8.15 Message: findFirst: discriminator

Synopsis

Answer the index of the first *element* of the receiver which causes `discriminator` to evaluate to *true* when the *element* is used as the parameter. Answer zero (0) if no such *element* is found.

Definition: <sequenceReadableCollection>

For each *element* of the receiver, `discriminator` is evaluated with the *element* as the parameter. Answer the index of the first *element* which results in an evaluation of *true*; no further *elements* are considered. If no such *element* exists in the receiver, answer 0.

The *elements* are traversed in the order specified by the `#do:` message for the receiver.

Parameters

`discriminator`	<monadicValuable>	uncaptured

Return Values

<integer> unspecified

Errors

If an evaluation of `discriminator` results in an object that does not conform to <boolean> .

If the *elements* of the receiver are inappropriate for use as arguments to `discriminator`.

5.7.8.16 Message: findLast: discriminator

Synopsis

Answer the index of the last *element* of the receiver which causes `discriminator` to evaluate to *true* when the *element* is used as the parameter. Answer zero (0) if no such *element* is found.

Definition: <sequenceReadableCollection>

For each *element* of the receiver, in reverse order starting with the last, `discriminator` is evaluated with the *element* as the parameter. Answer the index of the first *element* which results in an evaluation of *true*; no further *elements* are considered. Answer 0 if no such *element* is found in the receiver.

The *elements* are traversed in the order specified by the `#reverseDo:` message for the receiver.

Parameters

`discriminator` <monadicValuable> uncaptured

Return Values

<integer> unspecified

Errors

If an evaluation of `discriminator` results in an object that does not conform to <boolean> .

If the *elements* of the receiver are inappropriate for use as arguments to `discriminator`.

5.7.8.17 Message: first

Synopsis

Answer the first *element* of the receiver.

Definition: <sequenceReadableCollection>

Answer the *element* at index 1 in the receiver. The result is undefined if the receiver is empty (answers *true* to the `#isEmpty` message).

Return Values

<Object> state

Errors

none

5.7.8.18 Message: from: start to: stop do: operation

Synopsis

For those *elements* of the receiver between positions `start` and `stop`, inclusive, evaluate `operation` with each *element* of the receiver.

Definition: <sequenceReadableCollection>

For each index in the range `start` to `stop`, the `operation` is evaluated with the *element* at that index as its argument.

Parameters

`start`	<integer>	uncaptured
`stop`	<integer>	uncaptured
`operation`	<monadicValuable>	uncaptured

Return Values

UNSPECIFIED

Errors

If the *elements* of the receiver are inappropriate for use as arguments to `operation`.

`start` < 1

`stop` > receiver's size

5.7.8.19 Message: from: start to: stop keysAndValuesDo: operation

Synopsis

For those *elements* of the receiver between positions `start` and `stop`, inclusive, evaluate `operation` with an *element* of the receiver as the first argument and the *element's* position (index) as the second.

Definition: <sequenceReadableCollection>

For each index in the range `start` to `stop`, the `operation` is evaluated with the index as the first argument and the *element* at that index as the second argument.

Parameters

`start`	<integer>	uncaptured
`stop`	<integer>	uncaptured
`operation`	<dyadicValuable>	uncaptured

Return Values

UNSPECIFIED

Errors

If the *elements* of the receiver or its indices are inappropriate for use as arguments to `operation`.

`start` < 1

`stop` > receiver's size

5.7.8.20 Message: indexOf: target

Synopsis

Answer the index of the first *element* of the receiver which is *equivalent* to `target`. Answer zero (0) if no such *element* is found.

Definition: <sequenceReadableCollection>

Answer the index of the first *element* which is *equivalent* to `target`; no further *elements* are considered. Answer 0 if no such *element* exists in the receiver.

The *elements* are traversed in the order specified by the `#do:` message for the receiver.

Parameters

`target`	<Object>	uncaptured

Return Values

<integer> unspecified

Errors

none

5.7.8.21 Message: indexOf: target ifAbsent: exceptionHandler

Synopsis

Answer the index of the first *element* of the receiver which is *equivalent* to `target`. Answer the result of evaluating `exceptionHandler` with no parameters if no such *element* is found.

Definition: <sequenceReadableCollection>

Answer the index of the first *element* which is *equivalent* to `target`; no further *elements* are considered. Answer `exceptionHandler` evaluated with no parameters if no such *element* is found.

The *elements* are traversed in the order specified by the `#do:` message for the receiver.

Parameters

`target`	<Object>	uncaptured
`exceptionHandler`	<niladicValuable>	uncaptured

Return Values

<integer> unspecified

<Object> unspecified

Errors

none

5.7.8.22 Message: indexOfSubCollection: targetSequence startingAt: start

Synopsis

Answer the index of the first *element* of the receiver which is the start of a subsequence which matches `targetSequence`. Start searching at index `start` in the receiver. Answer 0 if no such subsequence is found.

Definition: <sequenceReadableCollection>

Each subsequence of the receiver starting at index `start` is checked for a match with `targetSequence`. To match, each *element* of a subsequence of the receiver must be *equivalent* to the corresponding *element* of `targetSequence`. Answer the index of the first *element* which begins a matching subsequence; no further subsequences are considered. Answer 0 if no such subsequence is found in the receiver, or if `targetSequence` is empty.

The *elements* are traversed in the order specified by the `#do:` message for the receiver.

Parameters

`targetSequence`	<sequencedReadableCollection>	uncaptured
`start`	<integer>	uncaptured

Return Values

<integer> unspecified

Errors

`start` < 1

`start` > the receiver's size

5.7.8.23 Message: indexOfSubCollection: targetSequence startingAt: start ifAbsent: exceptionHandler

Synopsis

Answer the index of the first *element* of the receiver which is the start of a subsequence which matches `targetSequence`. Start searching at index `start` in the receiver. Answer the result of evaluating `exceptionHandler` with no parameters if no such subsequence is found.

Definition: <sequenceReadableCollection>

Each subsequence of the receiver starting at index `start` is checked for a match with `targetSequence`. To match, each *element* of a subsequence of the receiver must be *equivalent* to the corresponding *element* of `targetSequence`. Answer the index of the first *element* which begins a matching subsequence; no further subsequences are considered. Answer the result of evaluating `exceptionHandler` with no parameters if no such subsequence is found or if `targetSequence` is empty.

The *elements* are traversed in the order specified by the `#do:` message for the receiver.

Parameters

`targetSequence`	<sequencedReadableCollection>	uncaptured
`start`	<integer>	uncaptured
`exceptionHandler`	<niladicValuable>	uncaptured

Return Values

<integer> unspecified

<Object> unspecified

Errors

`start` < 1

`start` > the receiver's size

5.7.8.24 Message: keysAndValuesDo: operation

Synopsis

Evaluate `operation` with the index of each *element* of the receiver, in order, together with the *element* itself.

Definition: <sequenceReadableCollection>

The `operation` is evaluated with the index of each *element* of the receiver as the first argument and the *element* itself as the second argument. Evaluation is in indexed order starting at 1. The first *element* is at index 1, the second at index 2, etc. The index of the last *element* is equal to the receiver's size.

Parameters

`operation`	<dyadicValuable>	uncaptured

Return Values

UNSPECIFIED

Errors

If the *elements* of the receiver are inappropriate for use as arguments to `operation`.

5.7.8.25 Message: last

Synopsis

Answer the last *element* of the receiver.

Definition: <sequenceReadableCollection>

Answer the last *element* of the receiver, the *element* at the index equal to the receiver's size. The result is unspecified if the receiver is empty (answers *true* to the `#isEmpty` message).

Return Values

<Object> state

Errors

none

5.7.8.26 Message: reverse

Synopsis

Answer a collection with the *elements* of the receiver arranged in reverse order.

Definition: <sequenceReadableCollection>

Answer a collection conforming to the same protocols as the receiver, but with its *elements* arranged in reverse order.

This operation is equivalent to:

1. Create a new collection which conforms to the same protocols as the receiver;

2. Traverse the *elements* of the receiver in the order specified by the `#reverseDo:` message, adding each *element* of the receiver to the new collection;

3. Answer the new collection.

Return Values

<RECEIVER> new

Errors

none

5.7.8.27 Message: reverseDo: operation

Synopsis

Evaluate `operation` with each *element* of the receiver in the reverse of the receiver's standard traversal order.

Definition: <sequenceReadableCollection>

For each *element* of the receiver, evaluate `operation` with the *element* as the parameter. The *elements* are traversed in the opposite order from the `#do:` message. Each *element* is visited exactly once.

Parameters

`operation` <monadicValuable> uncaptured

Return Values

UNSPECIFIED

Errors

If the *elements* of the receiver are inappropriate for use as arguments to `operation`.

5.7.8.28 Message: with: otherCollection do: operation

Synopsis

Evaluate `operation` with each *element* of the receiver and the corresponding *element* of `otherCollection` as parameters.

Definition: <sequenceReadableCollection>

For each *element* of the receiver and the corresponding *element* of `otherCollection`, evaluate `operation` with the receiver's *element* as the first parameter, and the *element* of `otherCollection` as the second parameter. The receiver and `otherCollection` must have the same size.

The *elements* of the receiver and `otherCollection` are traversed in indexed order starting at 1. The `operation` is first evaluated with the *elements* at index 1 in the two <sequencedReadableCollection>s, then index 2, etc.

Parameters

`otherCollection` <sequencedReadableCollection> uncaptured

`operation`	<dyadicValuable>	uncaptured

Return Values

UNSPECIFIED

Errors

If the *elements* of the receiver or the *elements* of `otherCollection` are inappropriate for use as arguments to `operation`.

If the receiver's size is not equal to the size of `otherCollection`.

5.7.9 Protocol: <Interval>

Conforms To

<sequencedReadableCollection>

Description

Represents a collection whose *elements* are numbers which form an arithmetic progression. Elements cannot be accessed externally.

Messages

,
collect:
copyFrom:to:
copyReplaceAll:with:
copyReplaceFrom:to:with:
copyReplaceFrom:to:withObject:
copyReplacing:withObject:
copyWith:
copyWithout:
reject:
reverse
select:

5.7.9.1 Message Refinement: , operand

Synopsis

Answer a new collection which is the concatenation of the receiver and `operand`.

Definition: <sequenceReadableCollection>

Answer a new collection containing all of the receiver's *elements* in their original order followed by all of the *elements* of `operand`, in their original order. The size of the new collection is equal to the sum of the sizes of the receiver and `operand`, as defined by the `#size` message.

Collections that enforce an ordering on their *elements* are permitted to refine this message to reorder the result.

Unless specifically refined, this message is defined to answer an instance of the same type as the receiver.

Refinement: <Interval>

Answer a collection containing the *elements* of `operand` appended to the *elements* of the receiver. The enumeration order defined by the `#do:` message is used. The return type is generalized to <sequencedReadableCollection>.

Parameters

`operand` <sequencedReadableCollection> uncaptured

Return Values

<sequencedReadableCollection> new

Errors

none

5.7.9.2 Message Refinement: collect: transformer

Synopsis

Answer a new collection constructed by gathering the results of evaluating `transformer` with each *element* of the receiver.

Definition: <collection>

For each element of the receiver, `transformer` is evaluated with the *element* as the parameter. The results of these evaluations are collected into a new collection.

The *elements* are traversed in the order specified by the `#do:` message for the receiver.

Unless specifically refined, this message is defined to answer an objects conforming to the same protocol as the receiver.

Refinement: <Interval>

The return type is generalized to <sequencedReadableCollection>.

Parameters

`transformer` <monadicValuable> uncaptured

Return Values

<sequencedReadableCollection> new

Errors

If the *elements* of the receiver are inappropriate for use as arguments to `transformer`.

5.7.9.3 Message Refinement: copyFrom: start to: stop

Synopsis

Answer a new collection containing all of the *elements* of the receiver between the indices `start` and `stop` inclusive. If `stop` < `start`, the result has a size of zero.

Definition: <sequenceReadableCollection>

Answer a new collection containing the specified range of *elements* of the receiver in their original order. The *element* at index `start` in the receiver is at index 1 in the new collection; the *element* at index `start+1` is at index 2, etc. If `stop` is less than `start`, then the new collection is empty. Otherwise, the size of the new collection is the maximum of (`stop` - `start` + 1) and 0.

The parameters `start` and `stop` must be positive.

Unless specifically refined, this message is defined to answer an instance of the same class as the receiver's class.

Refinement: <Interval>

The return type is generalized to <sequencedReadableCollection>.

Parameters

`start` <integer> uncaptured

`stop` <integer> uncaptured

Return Values

<sequencedReadableCollection> new

Errors

If `start` < 1 or `start` > self size.

If `stop` < 1 or `stop` > self size.

5.7.9.4 Message Refinement: copyReplaceAll: targetElements with: replacementElements

Synopsis

Answer a new collection in which all subsequences of *elements* in the receiver matching `targetElements` are replaced in the new collection by the *elements* in `replacementElements`.

Definition: <sequenceReadableCollection>

Answer a new collection with the *elements* of the receiver in their original order, except where a subsequence in the receiver matches `targetElements`. A subsequence in the receiver is said to match the *elements* of `targetElements` if:

1. They have the same number of *elements*.

2. For all indices of the subsequence, the *element* in the subsequence at a given index is *equivalent* to the *element* in `targetElements` at the same index.

Where a subsequence match is found, the *elements* from `replacementElements` are placed in the new collection instead.

Unless specifically refined, this message is defined to answer an instance of the same class as the receiver.

Collections that enforce an ordering on their *elements* are permitted to refine this message to reorder the result.

Refinement: <Interval>

The return type is generalized to <sequencedReadableCollection>.

Parameters

`targetElements`	<sequencedReadableCollection>	uncaptured
`replacementElements`	<sequencedReadableCollection>	unspecified

Return Values

<sequencedReadableCollection> new

Errors

If any of the *elements* in `replacementElements` is inappropriate for storage in instances of the result.

5.7.9.5 Message Refinement: copyReplaceFrom: start to: stop with: replacementElements

Synopsis

Answer a new collection, containing the same *elements* as the receiver, but with the *elements* in the receiver between `start` and `stop` inclusive replaced by the *elements* in `replacementElements`.

Definition: <sequenceReadableCollection>

This message can be used to insert, append, or replace. The size of `replacementElements` (as defined by `#size`) need not be the same as the number of *elements* being replaced. There are three cases:

1. If `stop` = `start` - 1 , and `start` is less than or equal to the size of the receiver, then the `replacementElements` are inserted between the *elements* at index `stop` and `start`. None of the receiver's *elements* are replaced.

2. If `stop` = the size of the receiver and `start` = `stop` + 1, then the operation is an append, and the `replacementElements` are placed at the end of the new collection.

3. Otherwise, the operation is a replacement, and the receiver's *elements* in the given range are replaced by the *elements* from `replacementElements`.

In all cases, the resulting collection consists of the receiver's *elements* from indices 1 to `start` - 1 in their original order, followed by the *elements* of `replacementElements`, followed by the remainder of the receiver's *elements* from index `stop` + 1 in their original order. The size of the result is the receiver's size - (`stop` - `start` + 1) + the `replacementElements` size.

The parameters `start` and `stop` must be positive.

Unless specifically refined, this message is defined to answer an instance of the same class as the receiver's class.

Collections that enforce an ordering on their *elements* are permitted to refine this message to reorder the result.

Refinement: <Interval>

The return type is generalized to <sequencedReadableCollection>.

Parameters

`start`	<integer>	uncaptured
`stop`	<integer>	uncaptured
`replacementElements`	<sequencedReadableCollection>	unspecified

Return Values

<sequencedReadableCollection> new

Errors

The *elements* in `replacementElements` are not suitable for storage in instances of the result.

5.7.9.6 Message Refinement: copyReplaceFrom: start to: stop withObject: replacementElement

Synopsis

Answer a new collection conforming to the same protocols as the receiver, in which the *elements* of the receiver between `start` and `stop` inclusive have been replaced with `replacementElement`.

Definition: <sequenceReadableCollection>

This message can be used to insert, append, or replace. There are three cases:

1. If `stop` = `start` - 1 , and `start` is less than or equal to the size of the receiver, then `replacementElement` is inserted between the *elements* at index `stop` and `start`. None of the receiver's *elements* are replaced.

2. If `stop` = the size of the receiver and `start` = `stop` + 1, then the operation is an append, and `replacementElement` is placed at the end of the new collection.

3. Otherwise, the operation is a replacement, and each of the receiver's *elements* in the given range is replaced by `replacementElement`.

The parameters `start` and `stop` must be non-negative.

Unless specifically refined, this message is defined to answer an instance of the same class as the receiver's class.

Collections that by definition enforce an ordering on their *elements* are permitted to refine this message to reorder the result.

Refinement: <Interval>

The return type is generalized to <sequencedReadableCollection>.

Parameters

`start`	<integer>	uncaptured
`stop`	<integer>	uncaptured

`replacementElement` <Object> captured

Return Values

<sequencedReadableCollection> new

Errors

none

5.7.9.7 Message Refinement: copyReplacing: targetElement withObject: replacementElement

Synopsis

Answer a new collection conforming to the same protocols as the receiver, in which any occurrences of `targetElement` are replaced by `replacementElement`.

Definition: <sequenceReadableCollection>

A new collection is created and initialized with the same *elements* as the receiver in the same order, except that any objects in the receiver which are *equivalent* to `targetElement` are replaced in the new collection by `replacementElement`.

Unless specifically refined, this message is defined to answer an instance of the same class as the receiver.

Collections that enforce an ordering on their *elements* are permitted to refine this message to reorder the result.

Refinement: <Interval>

The return type is generalized to <sequencedReadableCollection>.

Parameters

`targetElement` <Object> uncaptured

`replacementElement` <Object> captured

Return Values

<sequencedReadableCollection> new

Errors

none

5.7.9.8 Message Refinement: copyWith: newElement

Synopsis

Answer a new collection containing the same *elements* as the receiver, with `newElement` added.

Definition: <sequenceReadableCollection>

Answer a new collection with size one greater than the size of the receiver containing the *elements* of the receiver and `newElement` placed at the end.

Unless specifically refined, this message is defined to answer an instance of the same class as the receiver.

Collections that enforce an ordering on their *elements* are permitted to refine this message to reorder the result.

Refinement: <Interval>

The return type is generalized to <sequencedReadableCollection>.

Parameters

`newElement` <Object> captured

Return Values

<sequencedReadableCollection> new

Errors

none

5.7.9.9 Message Refinement: copyWithout: oldElement

Synopsis

Answer a new collection, containing the same *elements* as the receiver in their original order omitting any *elements equivalent* to `oldElement`.

Definition: <sequenceReadableCollection>

Answer a new collection with all of the *elements* of the receiver that are not *equivalent* to `oldElement`, in their original order.

Unless specifically refined, this message is defined to answer an instance of the same type as the receiver.

Refinement: <Interval>

The return type is generalized to <sequencedReadableCollection>.

Parameters

`oldElement` <Object> uncaptured

Return Values

<sequencedReadableCollection> new

Errors

none

5.7.9.10 Message Refinement: reject: discriminator

Synopsis

Answer a new collection which excludes the *elements* in the receiver which cause `discriminator` to evaluate to *true*.

Definition: <collection>

For each *element* of the receiver, `discriminator` is evaluated with the *element* as the parameter. Each *element* which causes `discriminator` to evaluate to *false* is added to the new collection.

The *elements* are traversed in the order specified by the `#do:` message for the receiver.

Unless specifically refined, this message is defined to answer an object conforming to the same protocol as the receiver.

Refinement: <Interval>

The return type is refined to <sequencedReadableCollection>.

Parameters

`discriminator` <monadicValuable> uncaptured

Return Values

<sequenceReadableCollection> new

Errors

If the *elements* of the receiver are inappropriate for use as arguments to `discriminator`.

If `discriminator` evaluates to an object that does not conform to the protocol <boolean> for any *element* of the receiver.

5.7.9.11 Message Refinement: reverse

Synopsis

Answer a collection with the *elements* of the receiver arranged in reverse order.

Definition: <sequenceReadableCollection>

Answer a collection conforming to the same protocols as the receiver, but with its *elements* arranged in reverse order.

This operation is equivalent to:

1. Create a new collection which conforms to the same protocols as the receiver;

2. Traverse the *elements* of the receiver in the order specified by the `#reverseDo:` message, adding each *element* of the receiver to the new collection;

3. Answer the new collection.

Refinement: <Interval>

The return type is generalized to <sequencedReadableCollection>.

Return Values

<sequenceReadableCollection> new

Errors

none

5.7.9.12 Message Refinement: select: discriminator

Synopsis

Answer a new collection which contains the *elements* in the receiver which cause `discriminator` to evaluate to *true*.

Definition: <collection>

For each *element* of the receiver, `discriminator` is evaluated with the *element* as the parameter. Each *element* which causes `discriminator` to evaluate to *true* is added to the new collection.

The *elements* are traversed in the order specified by the `#do:` message for the receiver.

Unless specifically refined, this message is defined to answer an objects conforming to the same protocol as the receiver.

Refinement: <Interval>

The return type is refined to <sequencedReadableCollection>.

Parameters

`discriminator` <monadicValuable> uncaptured

Return Values

<sequenceReadableCollection> new

Errors

If the *elements* of the receiver are inappropriate for use as arguments to `discriminator`.

If `discriminator` evaluates to an object that does not conform to the protocol <boolean> for any *element* of the receiver.

5.7.10 Protocol: <readableString>

Conforms To

<magnitude> <sequencedReadableCollection>

Description

Provides protocol for string operations such as copying, comparing, replacing, converting, indexing, and matching. All objects that conform to the protocol <readableString> are *comparable*.

Messages

,
<
<=
>
>=
asLowercase
asString
asSymbol
asUppercase
copyReplaceAll:with:
copyReplaceFrom:to:with:
copyReplacing:withObject:
copyWith:
sameAs:
subStrings:

5.7.10.1 Message Refinement: , operand

Synopsis

Answer a new collection which is the concatenation of the receiver and `operand`.

Definition: <sequenceReadableCollection>

Answer a new collection containing all of the receiver's elements in their original order followed by all of the elements of `operand`, in their original order. The size of the new collection is equal to the sum of the sizes of the receiver and `operand`, as defined by the `#size` message.

Collections that enforce an ordering on their elements are permitted to refine this message to reorder the result.

Unless specifically refined, this message is defined to answer an instance of the same type as the receiver.

Refinement: <readableString>

The parameter `operand` must be a <readableString>.

Parameters

`operand` <readableString> uncaptured

Return Values

<readableString> new

Errors

none

5.7.10.2 Message Refinement: < operand

Synopsis

Answer *true* if the receiver is less than `operand`. Answer *false* otherwise.

Definition: <magnitude>

Answer *true* if the receiver is less than `operand` with respect to the ordering defined for them. Answer *false* otherwise.

The result is undefined if the receiver and `operand` are not *comparable*.

The semantics of the natural ordering must be defined by refinement, which may also restrict the type of `operand`.

Refinement: <readableString>

Answer *true* if the receiver collates before `operand`, according to the implementation defined collating algorithm. Answer *false* otherwise.

Parameters

`operand`	<readableString>	uncaptured

Return Values

<boolean>	unspecified

Errors

none

5.7.10.3 Message Refinement: <= operand

Synopsis

Answer *true* if the receiver is less than or equal to `operand`. Answer *false* otherwise.

Definition: <magnitude>

Answer *true* if the receiver would answer *true* to either the #< or #= message with `operand` as the parameter. Answer *false* otherwise.

The result is undefined if the receiver and `operand` are not *comparable*.

Refinement: <readableString>

Answer *true* if the receiver answers true to either the #< or `#sameAs:` messages with `operand` as the parameter. Answer *false* otherwise.

Parameters

`operand`	<readableString>	uncaptured

Return Values

<boolean>	unspecified

Errors

none

5.7.10.4 Message Refinement: > operand

Synopsis

Answer *true* if the receiver is greater than `operand`. Answer *false* otherwise.

Definition: <magnitude>

Answer *true* if the receiver is greater than `operand` with respect to the natural ordering. Answer *false* otherwise.

The result is undefined if the receiver and `operand` are not *comparable*.

The semantics of the natural ordering must be defined by refinement, which may also restrict the type of `operand`.

Refinement: <readableString>

Answer *true* if the receiver collates after `operand`, according to the implementation defined collating algorithm. Answer *false* otherwise.

Parameters

`operand` <readableString> uncaptured

Return Values

<boolean> unspecified

Errors

none

5.7.10.5 Message Refinement: >= operand

Synopsis

Answer *true* if the receiver is greater than or equal to `operand`. Answer *false* otherwise.

Definition: <magnitude>

Answer *true* if the receiver answers *true* to either the #> or #= message with `operand` as the parameter. Answer *false* otherwise.

The result is undefined if the receiver and `operand` are not *comparable*.

Refinement: <readableString>

Answer *true* if the receiver answers true to either the #> or `#sameAs:` messages with `operand` as the parameter. Answer *false* otherwise.

Parameters

`operand` <readableString> uncaptured

Return Values

<boolean> unspecified

Errors

none

5.7.10.6 Message: asLowercase

Synopsis

Answer a new string which contains all of the elements of the receiver converted to their lower case equivalents.

Definition: <readableString>

Answer a new string which contains all of the elements of the receiver converted to their lower case equivalents. Individual element of the string are converted as if they were receivers of the message `#asLowercase`.

Return Values

<readableString> new

Errors

none

5.7.10.7 Message: asString

Synopsis

Answer a string containing the same characters as the receiver.

Definition: <readableString>

Answer a string containing the same characters as the receiver, in their original order.

Return Values

<readableString> unspecified

Errors

none

5.7.10.8 Message: asSymbol

Synopsis

Answer a symbol containing the same characters as the receiver.

Definition: <readableString>

Answer a symbol containing the same characters as the receiver, in their original order.

Return Values

<symbol> unspecified

Errors

none

5.7.10.9 Message: asUppercase

Synopsis

Answer a new string which contains all of the elements of the receiver converted to their upper case equivalents.

Definition: <readableString>

Answer a new string which contains all of the elements of the receiver converted to their upper case equivalents. Individual element of the string are converted as if they were receivers of the message `#asUppercase`.

Return Values

<readableString> new

Errors

none

5.7.10.10 Message Refinement: copyReplaceAll: targetElements with: replacementElements

Synopsis

Answer a new collection in which all subsequences of elements in the receiver matching `targetElements` are replaced in the new collection by the elements in `replacementElements`.

Definition: <sequenceReadableCollection>

Answer a new collection with the elements of the receiver in their original order, except where a subsequence in the receiver matches `targetElements`. A subsequence in the receiver is said to match the elements of `targetElements` if:

1. They have the same number of elements.

2. For all indices of the subsequence, the element in the subsequence at a given index is *equivalent* to the element in `targetElements` at the same index.

Where a subsequence match is found, the elements from `replacementElements` are placed in the new collection instead.

Unless specifically refined, this message is defined to answer an instance of the same class as the receiver.

Collections that enforce an ordering on their elements are permitted to refine this message to reorder the result.

Refinement: <readableString>

The elements of `targetElements` and `replacementElements` must conform to the protocol <character> and be valid elements for the result.

Parameters

`targetElements`	<sequenceReadableCollection>	uncaptured
`replacementElements`	<sequenceReadableCollection>	unspecified

Return Values

<readableString> new

Errors

none

5.7.10.11 Message Refinement: copyReplaceFrom: start to: stop with: replacementElements

Synopsis

Answer a new collection, containing the same elements as the receiver, but with the elements in the receiver between `start` and `stop` inclusive replaced by the elements in `replacementElements`.

Definition: <sequenceReadableCollection>

This message can be used to insert, append, or replace. The size of `replacementElements` (as defined by `#size`) need not be the same as the number of elements being replaced. There are three cases:

1. If `stop` = `start` - 1 , and `start` is less than or equal to the size of the receiver, then the `replacementElements` are inserted between the elements at index `stop` and `start`. None of the receiver's elements are replaced.

2. If `stop` = the size of the receiver and `start` = `stop` + 1, then the operation is an append, and the `replacementElements` are placed at the end of the new collection.

3. Otherwise, the operation is a replacement, and the receiver's elements in the given range are replaced by the elements from `replacementElements`.

In all cases, the resulting collection consists of the receiver's elements from indices 1 to `start` - 1 in their original order, followed by the elements of `replacementElements`, followed by the remainder of the receiver's elements from index `stop` + 1 in their original order. The size of the result is the receiver's size - (`stop` - `start` + 1) + the `replacementElements` size.

The parameters `start` and `stop` must be positive.

Unless specifically refined, this message is defined to answer an instance of the same class as the receiver's class.

Collections that enforce an ordering on their elements are permitted to refine this message to reorder the result.

Refinement: <readableString>

The elements of `replacementElements` must be characters.

Parameters

`start`	<integer>	uncaptured
`stop`	<integer>	uncaptured
`replacementElements`	<sequenceReadableCollection>	unspecified

Return Values

<readableString> new

Errors

none

5.7.10.12 Message Refinement: copyReplacing: targetElement withObject: replacementElement

Synopsis

Answer a new collection conforming to the same protocols as the receiver, in which any occurrences of `targetElement` are replaced by `replacementElement`.

Definition: <sequenceReadableCollection>

A new collection is created and initialized with the same elements as the receiver in the same order, except that any objects in the receiver which are *equivalent* to `targetElement` are replaced in the new collection by `replacementElement`.

Unless specifically refined, this message is defined to answer an instance of the same class as the receiver.

Collections that enforce an ordering on their elements are permitted to refine this message to reorder the result.

Refinement: <readableString>

The parameters `targetElement` and `replacementElement` must be characters.

Parameters

`targetElement`	<character>	uncaptured
`replacementElement`	<character>	captured

Return Values

<readableString> new

Errors

none

5.7.10.13 Message Refinement: copyWith: newElement

Synopsis

Answer a new collection containing the same elements as the receiver, with `newElement` added.

Definition: <sequenceReadableCollection>

Answer a new collection with size one greater than the size of the receiver containing the elements of the receiver and `newElement` placed at the end.

Unless specifically refined, this message is defined to answer an instance of the same class as the receiver.

Collections that enforce an ordering on their elements are permitted to refine this message to reorder the result.

Refinement: <readableString>

The parameter `newElement` must be characters.

Parameters

`newElement`	<character>	captured

Return Values

<readableString> new

Errors

none

5.7.10.14 Message: sameAs: operand

Synopsis

Answer *true* if the receiver collates the same as `operand`. Answer *false* otherwise.

Definition: <readableString>

Answer *true* if the receiver collates the same as `operand`, according to the implementation-defined collating algorithm. Answer *false* otherwise.

This message differs from the #= message because two strings which are not equal can collate the same, and because the receiver and `operand` do not need to conform to the same protocols.

Parameters

`operand` <readableString> uncaptured

Return Values

<boolean> unspecified

Errors

none

5.7.10.15 Message: subStrings: separators

Synopsis

Answer an array containing the substrings in the receiver separated by the elements of `separators`.

Definition: <readableString>

Answer an array of strings. Each element represents a group of characters separated by any of the characters in the list of `separators`.

Parameters

`separators` <sequencedReadableCollection> uncaptured

Return Values

<Array> unspecified

Errors

If the list of `separators` contains anything other than characters.

5.7.11 Protocol: <symbol>

Conforms To

<readableString>

Description

Represents an ordered, variable sized and immutable collection of characters. There is a unique object conforming to this protocol for every possible sequence of characters. Symbols are *identity objects*.

Messages

asString
asSymbol

5.7.11.1 Message Refinement: asString

Synopsis

Answer a string containing the same characters as the receiver.

Definition: <readableString>

Answer a string containing the same characters as the receiver, in their original order.

Refinement: <symbol>

Answer an object that is not identical to the receiver

Return Values

<readableString> unspecified

Errors

none

5.7.11.2 Message Refinement: asSymbol

Synopsis

Answer a symbol containing the same characters as the receiver.

Definition: <readableString>

Answer a symbol containing the same characters as the receiver, in their original order.

Refinement: <symbol>

Answer the receiver.

Return Values

<RECEIVER> unspecified

Errors

none

See Also

`#asString`

5.7.12 Protocol: <sequencedCollection>

Conforms To

<sequencedReadableCollection>

Description

Provides protocol for writing to an ordered collection of objects, whose *elements* can be accessed using external integer *keys*.

Messages

at:put:
atAll:put:
atAllPut:
replaceFrom:to:with:
replaceFrom:to:with:startingAt:
replaceFrom:to:withObject:

5.7.12.1 Message: at: index put: newElement

Synopsis

Replace the *element* in the receiver at `index` with `newElement`. Answer `newElement`.

Definition: <sequencedCollection>

This message sets one of the receiver's *elements* based on `index`. The `newElement` is stored at `index` in the receiver's *elements*, replacing any previously stored object. Subsequent retrievals at this index will answer `newElement`.

Parameters

`index`	<integer>	uncaptured
`newElement`	<Object>	captured

Return Values

<Object> state

Errors

If `index` < 0.

If `index` > the receiver's size.

If `newElement` does not conform to any *element type* restrictions of the receiver.

5.7.12.2 Message: atAll: indices put: newElement

Synopsis

Replace the *elements* in the receiver specified by `indices` with `newElement`.

Definition: <sequencedCollection>

The `newElement` is stored at each index in the receiver specified by the *elements* of the `indices` collection, replacing any previously stored objects at these indices. Subsequent retrievals at these indices will answer `newElement`.

This message is equivalent to storing `newElement` in the receiver at each index specified by `indices` using the `#at:put:` message for the receiver.

Parameters

`indices`	<collection>	uncaptured
`newElement`	<Object>	captured

Return Values

UNSPECIFIED

Errors

If any *element* of `indices` does not conform to <integer>.

If any *element* in `indices` is <= 0 or greater than the receiver's size.

If `newElement` does not conform to any *element type* restrictions of the receiver.

5.7.12.3 Message: atAllPut: newElement

Synopsis

Replace all the *elements* in the receiver with `newElement`.

Definition: <sequencedCollection>

The `newElement` is stored at each index in the receiver, replacing any previously stored objects.

This message is equivalent to storing `newElement` in the receiver at each index from 1 to the receiver's size using the `#at:put:` message for the receiver.

Parameters

`newElement`	<Object>	captured

Return Values

UNSPECIFIED

Errors

If `newElement` does not conform to any *element type* restrictions of the receiver.

5.7.12.4 Message: replaceFrom: start to: stop with: replacementElements

Synopsis

Replace the *elements* of the receiver between positions `start` and `stop` inclusive, with the *elements* of `replacementElements` in their original order. Answer the receiver.

Definition: <sequencedCollection>

The first *element* of `replacementElements` is stored in the receiver at position `start`, the second at position `start` + 1, etc. Any previously stored *elements* at these positions are replaced.

If the size of `replacementElements` is not equal to `stop` - `start` + 1, the result of sending this message is unspecified.

Parameters

`start`	<integer>	uncaptured
`stop`	<integer>	uncaptured
`replacementElements`	<sequencedReadableCollection>	unspecified

Return Values

UNSPECIFIED

Errors

If `start` < 1 or `start` > the receiver's size.

If `stop` < 1 or `stop` > the receiver's size.

If `replacementElements` size <> `stop` - `start` + 1.

5.7.12.5 Message: replaceFrom: start to: stop with: replacementElements startingAt: replacementStart

Synopsis

Replace the *elements* of the receiver between positions `start` and `stop` inclusive with the *elements* of `replacementElements`, in their original order, starting at position `replacementStart`. Answer the receiver.

Definition: <sequencedCollection>

The *element* at position `replacementStart` in `replacementElements` is stored in the receiver at position `start`; the *element* at `replacementStart` + 1 is stored at position `start` + 1; etc. Any previously stored *elements* at these positions in the receiver are replaced.

If the size of `replacementElements` is not equal to (`replacementStart` + `stop` - `start`), the result of sending this message is unspecified.

Parameters

`start`	<integer>	uncaptured
`stop`	<integer>	uncaptured
`replacementElements`	<sequencedReadableCollection>	unspecified
`replacementStart`	<integer>	uncaptured

Return Values

UNSPECIFIED

Errors

If `start` < 1 or `start` > the receiver's size.

If `stop` < 1 or `stop` > the receiver's size.

If `replacementStart` < 1 or `replacementStart` > `replacementElements` size.

If `replacementElements` size - `replacementStart` + 1 < `stop` - `start` + 1.

5.7.12.6 Message: replaceFrom: start to: stop withObject: replacementElement

Synopsis

Replace the *elements* of the receiver between `start` and `stop` inclusive with `replacementElement`. Answer the receiver.

Definition: <sequencedCollection>

Replace the *elements* of the receiver between `start` and `stop` inclusive with `replacementElement`. Answer the receiver.

Parameters

`start`	<integer>	uncaptured
`stop`	<integer>	uncaptured
`replacementElement`	<Object>	captured

Return Values

UNSPECIFIED

Errors

If `start` < 1 or `start` > the receiver's size.

If `stop` < 1 or `stop` > the receiver's size.

5.7.13 Protocol: <String>

Conforms To

<readableString> <sequencedCollection>

Description

Provides protocol for string operations such as copying, storing, comparing, replacing, converting, indexing, and matching. The *element type* of <String> is <Character>. The range of codePoints of characters that may be *elements* of a <String> is implementation defined.

Messages

asString

5.7.13.1 Message Refinement: asString

Synopsis

Answer a string containing the same characters as the receiver.

Definition: <readableString>

Answer a string containing the same characters as the receiver, in their original order.

Refinement: <String>

Answer the receiver.

Return Values

<String> unspecified

Errors

none

5.7.14 Protocol: <Array>

Conforms To

<sequencedCollection>

Description

Represents a keyed collection of objects which can be accessed externally using sequential integer *keys*. The index of the first *element* is one (1).

Messages

none

5.7.15 Protocol: <ByteArray>

Conforms To

<sequencedCollection>

Description

Represents a keyed collection whose *element type* is <integer> and is limited to the range 0 to 255, inclusive. The elements can be accessed externally using sequential integer *keys*. The index of the first *element* is one (1).

Messages

none

5.7.16 Protocol: <sequencedContractibleCollection>

Conforms To

<collection>

Description

Provides protocol for removing *elements* from an ordered collection of objects, whose *elements* can be accessed using external integer *keys*.

Messages

removeAtIndex:
removeFirst
removeLast

5.7.16.1 Message: removeAtIndex: index

Synopsis

Remove the *element* of the receiver at position `index`, and answer the removed *element*.

Definition: <sequenceContractibleCollection>

The *element* of the receiver which is at position `index` is removed from the receiver's *elements*. Answer the removed *element*.

`index` must be a positive integer less than or equal to the receiver's size.

Parameters

`index` <integer> uncaptured

Return Values

<Object> unspecified

Errors

If `index` is 0 or negative.

If `index` is greater than the receiver's size.

5.7.16.2 Message: removeFirst

Synopsis

Remove and answer the first *element* of the receiver.

Definition: <sequenceContractibleCollection>

The first *element* of the receiver is removed and answered. The *element* (if any) that was previously the second *element* in the traversal order now becomes the first, and the receiver has one fewer *elements*.

Return Values

<Object> state

Errors

The receiver is empty

5.7.16.3 Message: removeLast

Synopsis

Remove and answer the last *element* of the receiver.

Definition: <sequenceContractibleCollection>

The last *element* of the receiver is removed and answered. The *element* (if any) that was previously the second from last *element* in the traversal order now becomes the last, and the receiver has one fewer *elements*.

Return Values

<Object> state

Errors

The receiver is empty

5.7.17 Protocol: <SortedCollection>

Conforms To

<extensibleCollection> <sequencedContractibleCollection> <sequencedReadableCollection>

Description

Represents a variable sized collection of objects whose *elements* are ordered based on a sort order. The sort order is specified by a <dyadicValuable> called the *sort block*. *Elements* may be added, removed or inserted, and can be accessed using external integer *keys*.

Messages

,
add:
asSortedCollection
collect:
copyReplaceAll:with:
copyReplaceFrom:to:with:
copyReplaceFrom:to:withObject:
copyReplacing:withObject:
reverse
sortBlock
sortBlock:

5.7.17.1 Message Refinement: , operand

Synopsis

Answer a new collection which is the concatenation of the receiver and `operand`.

Definition: <sequenceReadableCollection>

Answer a new collection containing all of the receiver's *elements* in their original order followed by all of the *elements* of `operand`, in their original order. The size of the new collection is equal to the sum of the sizes of the receiver and `operand`, as defined by the `#size` message.

Collections that enforce an ordering on their *elements* are permitted to refine this message to reorder the result.

Unless specifically refined, this message is defined to answer an instance of the same type as the receiver.

Refinement: <SortedCollection>

Since the receiver sorts its *elements*, the result will also be sorted as defined by the receiver's *sort block*.

Parameters

`operand` <sequencedReadableCollection> uncaptured

Return Values

<SortedCollection> new

Errors

If the *elements* of `operand` cannot be sorted using receiver's *sort block*.

5.7.17.2 Message Refinement: add: newElement

Synopsis

Add `newElement` to the receiver's *elements*.

Definition: <extensibleCollection>

This message adds a `newElement` to the receiver. Unless specifically refined, the position of the `newElement` in the *element* traversal order is unspecified.

Conformant protocols may place restrictions on the type of objects that are valid elements. Unless otherwise specified, any object is acceptable.

Refinement: <SortedCollection>

Since the receiver maintains its *elements* in sorted order, the position of `newElement` will depend on the receiver's *sort block*.

Parameters

`newElement` <Object> captured

Return Values

UNSPECIFIED

Errors

If `newElement` cannot be sorted using receiver's *sort block*.

5.7.17.3 Message Refinement: asSortedCollection

Synopsis

Answer a sorted collection with the same *elements* as the receiver.

Definition: <collection>

Answer a sorted collection with the same *elements* as the receiver. The default *sort block* is used unless another *sort block* is specified in a refinement.

Refinement: <SortedCollection>

The receiver's *sort block* is used in the result.

Return Values

<SortedCollection> unspecified

Errors

none

5.7.17.4 Message Refinement: collect: transformer

Synopsis

Answer a new collection constructed by gathering the results of evaluating `transformer` with each *element* of the receiver.

Definition: <collection>

For each *element* of the receiver, `transformer` is evaluated with the *element* as the parameter. The results of these evaluations are collected into a new collection.

The elements are traversed in the order specified by the `#do:` message for the receiver.

Unless specifically refined, this message is defined to answer an objects conforming to the same protocol as the receiver.

Refinement: <SortedCollection>

Answer a <sequencedCollection>.

Parameters

`transformer` <monadicValuable> uncaptured

Return Values

<sequencedCollection> new

Errors

If the *elements* of the receiver are inappropriate for use as arguments to `transformer`.

If the result of evaluating the `transformer` is inappropriate for storage in the collection to be returned.

5.7.17.5 Message Refinement: copyReplaceAll: targetElements with: replacementElements

Synopsis

Answer a new collection in which all subsequences of *elements* in the receiver matching `targetElements` are replaced in the new collection by the *elements* in `replacementElements`.

Definition: <sequenceReadableCollection>

Answer a new collection with the *elements* of the receiver in their original order, except where a subsequence in the receiver matches `targetElements`. A subsequence in the receiver is said to match the *elements* of `targetElements` if:

1. They have the same number of *elements*.

2. For all indices of the subsequence, the *element* in the subsequence at a given index is *equivalent* to the *element* in `targetElements` at the same index.

Where a subsequence match is found, the *elements* from `replacementElements` are placed in the new collection instead.

Unless specifically refined, this message is defined to answer an instance of the same class as the receiver.

Collections that enforce an ordering on their *elements* are permitted to refine this message to reorder the result.

Refinement: <SortedCollection>

Since the receiver maintains its *elements* in sorted order, the positions of *elements* of `replacementElements` will depend on the receiver's *sort block*.

Parameters

`targetElements` <sequencedReadableCollection> uncaptured

`replacementElements` <sequencedReadableCollection> unspecified

Return Values

<SortedCollection> new

Errors

If any of the *elements* in `replacementElements` does not conform to any *element type* restrictions of instances of the receiver's class.

If the *elements* of `replacementElements` cannot be sorted using receiver's *sort block*.

5.7.17.6 Message Refinement: copyReplaceFrom: start to: stop with: replacementElements

Synopsis

Answer a new collection, containing the same *elements* as the receiver, but with the *elements* in the receiver between `start` and `stop` inclusive replaced by the *elements* in `replacementElements`.

Definition: <sequenceReadableCollection>

This message can be used to insert, append, or replace. The size of `replacementElements` (as defined by `#size`) need not be the same as the number of *elements* being replaced. There are three cases:

1. If `stop` = `start` - 1 , and `start` is less than or equal to the size of the receiver, then the `replacementElements` are inserted between the *elements* at index `stop` and `start`. None of the receiver's *elements* are replaced.

2. If `stop` = the size of the receiver and `start` = `stop` + 1, then the operation is an append, and the `replacementElements` are placed at the end of the new collection.

3. Otherwise, the operation is a replacement, and the receiver's *elements* in the given range are replaced by the *elements* from `replacementElements`.

In all cases, the resulting collection consists of the receiver's *elements* from indices 1 to `start` - 1 in their original order, followed by the *elements* of `replacementElements`, followed by the remainder of the receiver's *elements* from index `stop` + 1 in their original order. The size of the result is the receiver's size - (`stop` - `start` + 1) + the `replacementElements` size.

The parameters `start` and `stop` must be positive.

Unless specifically refined, this message is defined to answer an instance of the same class as the receiver's class.

Collections that enforce an ordering on their *elements* are permitted to refine this message to reorder the result.

Refinement: <SortedCollection>

Since the receiver maintains its *elements* in sorted order, the positions of elements of `replacementElements` will depend on the receiver's *sort block*.

Parameters

`start`	<integer>	uncaptured
`stop`	<integer>	uncaptured
`replacementElements`	<sequencedReadableCollection>	unspecified

Return Values

<SortedCollection>	new

Errors

The *elements* in `replacementElements` are not suitable for storage in instances of the receiver's class.

`start` > receiver's size + 1

`start` < 1

`stop` > receiver's size

`stop` < `start` - 1

If the *elements* of `replacementElements` cannot be sorted using receiver's *sort block*.

5.7.17.7 Message Refinement: copyReplaceFrom: start to: stop withObject: replacementElement

Synopsis

Answer a new collection conforming to the same protocols as the receiver, in which the *elements* of the receiver between `start` and `stop` inclusive have been replaced with `replacementElement`.

Definition: <sequenceReadableCollection>

This message can be used to insert, append, or replace. There are three cases:

1. If `stop` = `start` - 1 , and `start` is less than or equal to the size of the receiver, then `replacementElement` is inserted between the *elements* at index `stop` and `start`. None of the receiver's *elements* are replaced.

2. If `stop` = the size of the receiver and `start` = `stop` + 1, then the operation is an append, and `replacementElement` is placed at the end of the new collection.

3. Otherwise, the operation is a replacement, and each of the receiver's *elements* in the given range is replaced by `replacementElement`.

The parameters `start` and `stop` must be non-negative.

Unless specifically refined, this message is defined to answer an instance of the same class as the receiver's class.

Collections that by definition enforce an ordering on their *elements* are permitted to refine this message to reorder the result.

Refinement: <SortedCollection>

Since the receiver maintains its *elements* in sorted order, the position(s) occupied by `replacementElement` will depend on the receiver's *sort block*.

Parameters

`start`	<integer>	uncaptured
`stop`	<integer>	uncaptured
`replacementElement`	<Object>	captured

Return Values

<SortedCollection>	new

Errors

The `replacementElement` is not suitable for storage in instances of the receiver's class.

`start` > receiver's size + 1

`start` < 1

`stop` > receiver's size

`stop` < `start` - 1

If `replacementElement` cannot be sorted using receiver's *sort block*.

5.7.17.8 Message Refinement: copyReplacing: targetElement withObject: replacementElement

Synopsis

Answer a new collection conforming to the same protocols as the receiver, in which any occurrences of `targetElement` are replaced by `replacementElement`.

Definition: <sequenceReadableCollection>

A new collection is created and initialized with the same *elements* as the receiver in the same order, except that any objects in the receiver which are *equivalent* to `targetElement` are replaced in the new collection by `replacementElement`.

Unless specifically refined, this message is defined to answer an instance of the same class as the receiver.

Collections that enforce an ordering on their *elements* are permitted to refine this message to reorder the result.

Refinement: <SortedCollection>

Since the receiver maintains its *elements* in sorted order, the position occupied by `replacementElement` will depend on the receiver's *sort block*.

Parameters

`targetElement`	<Object>	uncaptured
`replacementElement`	<Object>	captured

Return Values

<SortedCollection>	new

Errors

If the `replacementElement` is inappropriate for storage in instances of the receiver's class.

If `replacementElement` cannot be sorted using receiver's *sort block*.

5.7.17.9 Message Refinement: reverse

Synopsis

Answer a collection with the *elements* of the receiver arranged in reverse order.

Definition: <sequenceReadableCollection>

Answer a collection conforming to the same protocols as the receiver, but with its *elements* arranged in reverse order.

This operation is equivalent to:

1. Create a new collection which conforms to the same protocols as the receiver;

2. Traverse the *elements* of the receiver in the order specified by the `#reverseDo:` message, adding each *element* of the receiver to the new collection;

3. Answer the new collection.

Refinement: <SortedCollection>

Answer a <sequencedReadableCollection>.

Return Values

<sequencedReadableCollection>	new

Errors

none

5.7.17.10 Message: sortBlock

Synopsis

Answer the receiver's *sort block*.

Definition: <SortedCollection>

Answer the receiver's *sort block*. The *sort block* is defined by the `#sortBlock:` message.

Return Values

<dyadicValuable>	state

Errors

none

5.7.17.11 Message: sortBlock: discriminator

Synopsis

Set the receiver's *sort block* to `discriminator`.

Definition: <SortedCollection>

This message defines the *sort block* used to specify the receiver's ordering criteria. The sortBlock is a 2-parameter <block>, which when evaluated with any two *elements* in the receiver, answers *true* if the first parameter should be ordered before the second parameter, and *false* otherwise. The *sort block* must obey the following properties:

1. Given the same 2 parameters, the *sort block* must answer the same result.

2. The *sort block* must obey transitivity. For example, if a is before b, and b is before c, then a must be before c.

The receiver's *sort block* is set to `discriminator`, and the *elements* are re-sorted.

Parameters

`discriminator` <dyadicValuable> captured

Return Values

<SortedCollection> receiver

Errors

If the *elements* of the receiver cannot be sorted using the `discriminator`.

5.7.18 Protocol: <OrderedCollection>

Conforms To

<extensibleCollection> <sequencedContractibleCollection> <sequencedCollection>

Description

Represents an ordered, variable sized collection of objects. *Elements* may be added, removed or inserted, and can be accessed using external integer *keys*.

Messages

add:
add:after:
add:afterIndex:
add:before:
add:beforeIndex:
addAll:after:
addAll:afterIndex:
addAll:before:
addAll:beforeIndex:
addAllFirst:
addAllLast:
addFirst:
addLast:

5.7.18.1 Message Refinement: add: newElement

Synopsis

Add `newElement` to the receiver's *elements*.

Definition: <extensibleCollection>

This message adds a `newElement` to the receiver. Unless specifically refined, the position of the `newElement` in the *element* traversal order is unspecified.

Conformant protocols may place restrictions on the type of objects that are valid *elements*. Unless otherwise specified, any object is acceptable.

Refinement: <OrderedCollection>

The `newElement` is added to the end of the receiver's *elements* so that it becomes the last element in the traversal order. This message is equivalent to `#addLast:` for the receiver with `newElement` as the parameter.

Parameters

`newElement`	<Object>	captured

Return Values

UNSPECIFIED

Errors

none

5.7.18.2 Message: add: newElement after: target

Synopsis

Add `newElement` to the receiver immediately following the first *element* which is *equivalent* to `target`.

Definition: <OrderedCollection>

Add `newElement` to the receiver immediately following the first *element* which is *equivalent* to `target`. An *element* immediately follows another if its index is one greater than that of the other. The order used to determine which of the receiver's *elements* is the first to equal `target` is the traversal order defined by `#do:` for the receiver.

If the receiver does not include `target`, the operation fails.

Parameters

`newElement`	<Object>	captured
`target`	<Object>	uncaptured

Return Values

<Object> state

Errors

If there is no *element* in the receiver which is *equivalent* to `target`.

5.7.18.3 Message: add: newElement afterIndex: index

Synopsis

Add `newElement` to the receiver immediately following the *element* at position `index`.

Definition: <OrderedCollection>

Add `newElement` to the receiver immediately following the *element* at position `index`. `newElement` is inserted at position `index` + 1. If `index` is equal to 0, `newElement` becomes the first *element* of the receiver.

Parameters

`newElement`	<Object>	captured
`index`	<integer>	uncaptured

Return Values

<Object> state

Errors

If `index` < 0.

If `index` > receiver's size.

5.7.18.4 Message: add: newElement before: target

Synopsis

Add `newElement` to the receiver immediately before the first *element* which is *equivalent* to `target`.

Definition: <OrderedCollection>

Add `newElement` to the receiver immediately before the first *element* which is *equivalent* to `target`. An *element* immediately precedes another if its index is one less than that of the other. The order used to determine which of the receiver's *elements* is the first to equal `target` in the traversal order defined by `#do:` for the receiver.

If the receiver does not include `target`, the operation fails.

Parameters

`newElement`	<Object>	captured
`target`	<Object>	uncaptured

Return Values

<Object> state

Errors

If there is no *element* in the receiver which is *equivalent* to `target`.

If the *element* which is equal to `target` is the last *element* in the receiver.

5.7.18.5 Message: add: newElement beforeIndex: index

Synopsis

Add `newElement` to the receiver immediately before the *element* at position `index`.

Definition: <OrderedCollection>

Add `newElement` to the receiver immediately before the *element* at position `index` in the receiver. If `index` equals the receiver's size plus 1 `newElement` will be inserted at the end of the receiver.

The parameter `index` must be a positive integer less than or equal to the receiver's size plus 1.

Parameters

`newElement`	<Object>	captured
`index`	<integer>	uncaptured

Return Values

<Object> state

Errors

If `index` <=0.

If `index` > receiver's size + 1.

5.7.18.6 Message: addAll: newElements after: target

Synopsis

Add each *element* of `newElements` to the receiver immediately after the first *element* in the receiver which is *equivalent* to `target`. Answer `newElements`.

Definition: <OrderedCollection>

Add the *elements* of `newElements` to the receiver in the traversal order defined by `#do:` for `newElements`. The new *elements* are inserted in the receiver immediately after the first *element* in the receiver which is *equivalent* to `target`.

An *element* immediately follows another if its index is one greater than that of the other. The order used to determine which of the receiver's *elements* is the first to equal `target` is the traversal order defined by `#do:` for the receiver.

If the receiver does not include `target`, the operation fails.

Parameters

`newElements`	<collection>	unspecified
`target`	<Object>	uncaptured

Return Values

UNSPECIFIED

Errors

If there is no *element* in the receiver which is *equivalent* to `target`.

5.7.18.7 Message: addAll: newElements afterIndex: index

Synopsis

Insert the *elements* of `newElements` in the receiver immediately after the *element* at position `index`. Answer `newElements`.

Definition: <OrderedCollection>

Add the *elements* of `newElements` to the receiver in the traversal order defined by `#do:` for `newElements`. The new *elements* are inserted in the receiver immediately after the *element* in the receiver at position `index`. If `index` is equal to 0, `newElements` are inserted at the beginning of the receiver.

The parameter `index` must be a non-negative integer less than or equal to the receiver's size.

Parameters

`newElements`	<collection>	unspecified
`index`	<integer>	uncaptured

Return Values

UNSPECIFIED

Errors

If `index` < 0.

If `index` > receiver's size.

5.7.18.8 Message: addAll: newElements before: target

Synopsis

Add each *element* of `newElements` to the receiver immediately before the first *element* in the receiver which is *equivalent* to `target`. Answer `newElements`.

Definition: <OrderedCollection>

Add the *elements* of `newElements` to the receiver in the traversal order defined by `#do:` for `newElements`. The new *elements* are inserted in the receiver immediately before the first *element* in the receiver which is *equivalent* to `target`.

An *element* immediately follows another if its index is one greater than that of the other. The order used to determine which of the receiver's *elements* is the first to equal `target` is the traversal order defined by `#do:` for the receiver.

If the receiver does not include `target`, the operation fails.

Parameters

`newElements`	<collection>	unspecified
`target`	<Object>	uncaptured

Return Values

UNSPECIFIED

Errors:

If there is no *element* in the receiver which is *equivalent* to `target`.

5.7.18.9 Message: addAll: newElements beforeIndex: index

Synopsis

Insert the *elements* of `newElements` in the receiver immediately before the *element* at position `index`. Answer `newElements`.

Definition: <OrderedCollection>

Add the *elements* of `newElements` to the receiver in the traversal order defined by `#do:` for `newElements`. The new *elements* are inserted in the receiver immediately before the *element* in the receiver at position `index`. If `index` equals the receiver's size plus 1 `newElements` will be inserted at the end of the receiver.

The parameter `index` must be a positive integer less than or equal to the receiver's size plus 1.

Parameters

`newElements`	<collection>	unspecified
`index`	<integer>	uncaptured

Return Values

UNSPECIFIED

Errors

If `index` <=0.

If `index` > receiver's size + 1.

5.7.18.10 Message: addAllFirst: newElements

Synopsis

Add each *element* of `newElements` to the beginning of the receiver's *elements*. Answer `newElements`.

Definition: <OrderedCollection>

This message is used to iteratively add each *element* of a given collection to the beginning of the receiver's *elements*.

The operation is equivalent to adding each successive *element* of `newElements` to the receiver using the `#addFirst:` message with the *element* as the parameter, where the `newElements` are traversed in the order specified by the `#reverseDo:` message for `newElements`.

Parameters

`newElements`	<sequencedCollection>	unspecified

Return Values

UNSPECIFIED

Errors

none

5.7.18.11 Message: addAllLast: newElements

Synopsis

Add each *element* of `newElements` to the end of the receiver's *elements*. Answer `newElements`.

Definition: <OrderedCollection>

This message is used to iteratively add each *element* of a given collection to the end of the receiver's *elements*.

The operation is equivalent to adding each successive *element* of `newElements` to the receiver using the `#addLast:` message with the *element* as the parameter, where the `newElements` are traversed in the order specified by the `#do:` message for `newElements`.

Parameters

UNSPECIFIED

Return Values

<sequencedCollection> parameter

Errors

none

5.7.18.12 Message: addFirst: newElement

Synopsis

Add `newElement` to the beginning of the receiver's *elements*. Answer `newElement`.

Definition: <OrderedCollection>

The `newElement` is added to the beginning of the receiver's *elements* so that it becomes the first *element* in the traversal order.

Parameters

`newElement` <Object> captured

Return Values

UNSPECIFIED

Errors

none

5.7.18.13 Message: addLast: newElement

Synopsis

Add `newElement` to the end of the receiver's *elements*. Answer `newElement`.

Definition: <OrderedCollection>

The `newElement` is added to the end of the receiver's *elements* so that it becomes the last *element* in the traversal order.

Parameters

`newElement` <Object> captured

Return Values

UNSPECIFIED

Errors

none

5.7.19 Protocol: <Interval factory>

Conforms To

<Object>

Description

Represents protocol for creating a collection whose *elements* are numbers which form an arithmetic progression.

Standard Globals

`Interval`	Conforms to the protocol <Interval factory>. Its language element type is unspecified. This is a factory and discriminator for collections that conform to <Interval>.

Messages

from:to:
from:to:by:

5.7.19.1 Message: from: start to: stop

Synopsis

Answer an interval which represents an arithmetic progression from `start` to `stop` in increments of 1.

Definition: <Interval factory>

Answer an interval which represents an arithmetic progression from `start` to `stop`, using the increment 1 to compute each successive *element*. The *elements* are numbers which have the same type as `start`. Note that `stop` may not be the last *element* in the sequence; the last *element* is given by the formula

```
start + ((stop - start) // 1)
```

The interval answered will be empty (it will answer 0 to the `#size` message) if `start` > `stop`

Parameters

`start`	<number>	unspecified
`stop`	<number>	unspecified

Return Values

<Interval>	unspecified

Errors

none

5.7.19.2 Message: from: start to: stop by: step

Synopsis

Answer an interval which represents an arithmetic progression from `start` to `stop` in increments of `step`.

Definition: <Interval factory>

Answer an interval which represents an arithmetic progression from `start` to `stop`, using the increment `step` to compute each successive *element*. The value of `step` can be positive or

negative, but it must be non-zero. The *elements* are numbers which have the most general type of `start` and `step`. Note that `stop` is not necessarily an *element* in the sequence; the last element is given by the formula

```
(((stop - start) // step) * step) + start
```

The interval answered will be empty (it will answer 0 to the `#size` message) if:

`start` < `stop` and `step` < 0, or

`start` > `stop` and `step` > 0.

Parameters

`start`	<number>	unspecified
`stop`	<number>	unspecified
`step`	<number>	unspecified

Return Values

<Interval>	unspecified

Errors

`step` = 0

5.7.20 Protocol: <collection factory>

Conforms To

<instantiator>

Description

Provides protocol for creating a collection of objects. A collection can be fixed or variable sized, ordered or unordered, and its *elements* may or may not be accessible by external *keys*.

Messages

new
new:

5.7.20.1 Message Refinement: new

Synopsis

Create a new object.

Definition: <instantiator>

Return a newly created object initialized to a standard initial state.

Refinement: <collection factory>

This message has the same effect as sending the message `#new:` with the argument `0`.

Return Values

<Collection>	new

Errors

none

5.7.20.2 Message: new: count

Synopsis

Create a new collection. The parameter `count` constrains the number of *elements* in the result.

Definition: <collection factory>

Return a new collection that has space for at least `count` *elements*.

Conforming protocols may refine this message. In particular, the effect of the parameter `count` should be specified in refinements. It can be used to specify the exact number of *elements*, the minimum number, or in some cases can even be interpreted as a hint from the programmer, with no guarantee that the requested number of instance variables will actually be allocated.

Unless otherwise stated the initial values of *elements*, if any, of the new collection are unspecified.

Parameters

`count` <integer> unspecified

Return Values

<collection> new

Errors

none

5.7.21 Protocol: <Dictionary factory>

Conforms To

<collection factory>

Description

This protocol defines the behavior of objects that can be used to create objects that conform to the protocol <Dictionary>.

Standard Globals

`Dictionary`	Conforms to the protocol <Dictionary factory>. Its language element type is unspecified. This is a factory and discriminator for collections that conform to <Dictionary>.

Messages

new
new:
withAll:

5.7.21.1 Message Refinement: new

Synopsis

Create a new object.

Definition: <instantiator>

Return a newly created object initialized to a standard initial state.

Refinement: <collection factory>

This message has the same effect as sending the message `#new:` with the argument `0`, and will return an empty collection.

Refinement: <Dictionary factory>

Return a new <Dictionary> that is optimized to store an implementation defined number of *elements*. The new collection initially contains no elements.

Return Values

<Dictionary> new

Errors

none

5.7.21.2 Message Refinement: new: count

Synopsis

Create a new collection. The parameter `count` constrains the number of *elements* in the result.

Definition: <collection factory>

Return a new collection that has space for at least `count` *elements*.

Conforming protocols may refine this message. In particular, the effect of the parameter `count` should be specified in refinements. It can be used to specify the exact number of *elements*, the minimum number, or in some cases can even be interpreted as a hint from the programmer, with no guarantee that the requested number of instance variables will actually be allocated.

Unless otherwise stated the initial values of *elements* of the new instance of the receiver are unspecified.

Refinement: <Dictionary factory>

The parameter `count` represents a hint for space allocation. The new collection is to optimized to contain `count` *elements*. The new collection initially contains no elements.

The new collection conforms to the protocol <Dictionary>.

Parameters

`count` <integer> unspecified

Return Values

<Dictionary> new

Errors

none

5.7.21.3 Message: withAll: newElements

Synopsis

Create a collection containing all the *elements* of `newElements`.

Definition: <Dictionary factory>

Return a new collection whose *elements* are the *elements* of `newElements`. The effect is the same as evaluating `Dictionary new addAll: newElements; yourself.`

Parameters

`newElements` <abstractDictionary> unspecified

Return Values

<Dictionary> new

Errors

none

5.7.22 Protocol: <IdentityDictionary factory>

Conforms To

<abstractDictionary factory>

Description

This protocol defines the behavior of objects that can be used to create objects that conform to the protocol <IdentityDictionary>.

Standard Globals

`IdentityDictionary`	Conforms to the protocol <IdentityDictionary factory>. Its language element type is unspecified. This is a factory and discriminator for collections that conform to <IdentityDictionary>.

Messages

new
new:
withAll:

5.7.22.1 Message Refinement: new

Synopsis

Create a new object.

Definition: <instantiator>

Return a newly created object initialized to a standard initial state.

Refinement: <collection factory>

This message has the same effect as sending the message `#new:` with the argument `0`, and will return an empty collection.

Refinement: <IdentityDictionary factory>

Return a new <IdentityDictionary> that is optimized to store an implementation defined number of *elements*. The new collection initially contains no elements.

Return Values

<IdentityDictionary>	new

Errors

none

5.7.22.2 Message Refinement: new: count

Synopsis

Create a new collection. The parameter `count` constrains the number of *elements* in the result.

Definition: <collection factory>

Return a new collection that has space for at least `count` *elements*.

Conforming protocols may refine this message. In particular, the effect of the parameter `count` should be specified in refinements. It can be used to specify the exact number of *elements*, the minimum number, or in some cases can even be interpreted as a hint from the programmer, with no guarantee that the requested number of instance variables will actually be allocated.

Unless otherwise stated the initial values of *elements* of the new instance of the receiver are unspecified.

Refinement: <IdentityDictionary factory>

The parameter `count` represents a hint for space allocation. The new collection is to optimized to contain `count` *elements*. The new collection initially contains no elements.

The new collection conforms to the protocol <IdentityDictionary>.

Parameters

`count` <integer> unspecified

Return Values

<IdentityDictionary> new

Errors

none

5.7.22.3 Message: withAll: newElements

Synopsis

Create a collection containing all the *elements* of `newElements`.

Definition: <IdentityDictionary factory>

Return a new collection whose *elements* are the *elements* of `newElements`. The effect is the same as evaluating `IdentityDictionary new addAll: newElements; yourself`.

Parameters

`newElements` <abstractDictionary> unspecified

Return Values

<IdentityDictionary> new

Errors

none

5.7.23 Protocol: <initializableCollection factory>

Conforms To

<collection factory>

Description

This protocol defines the behavior of objects that can be used to create non-empty collections.

Messages

with:
with:with:
with:with:with
with:with:with:with:
withAll:

5.7.23.1 Message: with: element1
Message: with: element1 with: element2

Message: with: element1 with: element2 with: element3
Message: with: element1 with: element2 with: element3 with: element4

Synopsis

Create a collection initially containing the argument elements

Definition: <initializableCollection factory>

Return a new collection containing a number of elements equal to the number of arguments to this message. The collection contains the arguments as its *elements*.

Conforming protocols may impose restrictions on the values of the arguments and hence the *element types*.

Parameters

`element1`	<Object>	captured
`element2`	<Object>	captured
`element3`	<Object>	captured
`element4`	<Object>	captured

Return Values

<collection>	new

Errors

If any of the arguments do not meet the *element type* constraints of the result object

5.7.23.2 Message: withAll: newElements

Synopsis

Create a collection containing all the *elements* of `newElements`.

Definition: <initializableCollection factory>

Return a new collection whose *elements* are the *elements* of `newElements`.

Conforming protocols may impose restrictions on the values of `newElements`.

Parameters

`newElements`	<collection>	unspecified

Return Values

<collection>	new

Errors

If any of the elements in `newElements` do not meet the *element type* constraints of the result object

5.7.24 Protocol: <Array factory>

Conforms To

<initializableCollection factory>

Description

This protocol defines the behavior of objects that can be used to create objects that conform to <Array>. These objects are created with a specified size. If element values are not explicitly provided they default to nil.

Standard Globals

`Array`	Conforms to the protocol <Array factory>. Its language element type is unspecified. This is a factory and discriminator for collections that conform to <Array>.

Messages

new
new:
with:
with:with:
with:with:with
with:with:with:with:
withAll:

5.7.24.1 Message Refinement: new

Synopsis

Create a new object.

Definition: <instantiator>

Return a newly created object initialized to a standard initial state.

Refinement: <collection factory>

This message has the same effect as sending the message `#new:` with the argument `0`, and will return an empty collection.

Refinement: <Array factory>

Create a new <Array> that contains no elements.

Return Values

<Array> new

Errors

none

5.7.24.2 Message Refinement: new: count

Synopsis

Create a new collection. The parameter `count` constrains the number of *elements* in the result.

Definition: <collection factory>

Return a new collection that has space for at least `count` *elements*.

Conforming protocols may refine this message. In particular, the effect of the parameter `count` should be specified in refinements. It can be used to specify the exact number of *elements*, the minimum number, or in some cases can even be interpreted as a hint from the programmer, with no guarantee that the requested number of instance variables will actually be allocated.

Unless otherwise stated the initial values of *elements* of the new instance of the receiver are unspecified.

Refinement: <Array factory>

The parameter `count` specifies the size of the receiver. The initial value of each *element* of the new instance of the receiver is *nil*. The new collections conforms to the protocol <Array>.

Parameters

`count`	<integer>	unspecified

Return Values

<Array> new

Errors

`count<0`

5.7.24.3 Message Refinement: with: element1
Message Refinement: with: element1 with: element2
Message Refinement: with: element1 with: element2 with: element3
Message Refinement: with: element1 with: element2 with: element3 with: element4

Synopsis

Create a collection initially containing the argument elements

Definition: <initializableCollection factory>

Return a new collection containing a number of elements equal to the number of arguments to this message. The collection contains the arguments as its *elements*.

Conforming protocols may impose restrictions on the values of the arguments and hence the *element types*.

Refinement: <Array factory>

The first argument is at index position 1, the second argument is at index position 2, and so on...

Parameters

`element1`	<Object>	captured
`element2`	<Object>	captured
`element3`	<Object>	captured
`element4`	<Object>	captured

Return Values

<Array> new

Errors

If any of the arguments do not meet the *element type* constraints of the result object

5.7.24.4 Message Refinement: withAll: newElements

Synopsis

Create a collection containing only the *elements* of `newElements`.

Definition: <initializableCollection factory>

Return a new collection whose *elements* are the *elements* of `newElements`.

Conforming protocols may impose restrictions on the values of `newElements`.

Refinement: <Array factory>

If the elements of `newElements` are ordered then their ordering establishing their index positions in the new collection.

Parameters

`newElements`	<collection>	unspecified

Return Values

<Array> new

Errors

If any of the elements of `newElements` do not meet the *element type* constraints of the result object

5.7.25 Protocol: <Bag factory>

Conforms To

<initializableCollection factory>

Description

This protocol defines the behavior of objects that can be used to create objects that conform to the protocol <Bag>.

Standard Globals

`Bag`	Conforms to the protocol <Bag factory>. Its language element type is unspecified. This is a factory and discriminator for collections that conform to <Bag>.

Messages

new
new:
with:
with:with:
with:with:with
with:with:with:with:
withAll:

5.7.25.1 Message Refinement: new

Synopsis

Create a new object.

Definition: <instantiator>

Return a newly created object initialized to a standard initial state.

Refinement: <collection factory>

This message has the same effect as sending the message `#new:` with the argument `0`, and will return an empty collection.

Refinement: <Bag factory>

Return a new <Bag> that is optimized to store an implementation defined number of *elements*. The new collection initially contains no elements.

Return Values

<Bag> new

Errors

none

5.7.25.2 Message Refinement: new: count

Synopsis

Create a new collection. The parameter `count` constrains the number of *elements* in the result.

Definition: <collection factory>

Return a new collection that has space for at least `count` *elements*.

Conforming protocols may refine this message. In particular, the effect of the parameter `count` should be specified in refinements. It can be used to specify the exact number of *elements*, the minimum number, or in some cases can even be interpreted as a hint from the programmer, with no guarantee that the requested number of instance variables will actually be allocated.

Unless otherwise stated the initial values of *elements* of the new instance of the receiver are unspecified.

Refinement: <Bag factory>

The parameter `count` represents a hint to the implementation as to the likely number of elements that may be added to the new collection. The new collection initially contains no elements.

The new collections conforms to the protocol <Bag>.

Parameters

`count`	<integer>	unspecified

Return Values

<Bag> new

Errors

none

5.7.25.3 Message Refinement: with: element1
Message Refinement: with: element1 with: element2
Message Refinement: with: element1 with: element2 with: element3
Message Refinement: with: element1 with: element2 with: element3 with: element4

Synopsis

Create a collection initially containing the argument elements

Definition: <initializableCollection factory>

Return a new collection containing a number of elements equal to the number of arguments to this message. The collection contains the arguments as its *elements*.

Conforming protocols may impose restrictions on the values of the arguments and hence the *element types*.

Refinement: <Bag factory>

The result is undefined if any of the arguments are *nil*.

Parameters

`element1`	<Object>	captured
`element2`	<Object>	captured
`element3`	<Object>	captured
`element4`	<Object>	captured

Return Values

<Bag> new

Errors

If any of the arguments do not meet the *element type* constraints of the result object

5.7.25.4 Message Refinement: withAll: newElements

Synopsis

Create a collection containing only the *elements* of `newElements`.

Definition: <initializableCollection factory>

Return a new collection whose *elements* are the *elements* of `newElements`.

Conforming protocols may impose restrictions on the values of `newElements`.

Refinement: <Bag factory>

The result is unspecified if `newElements` contains *nil*.

Parameters

`newElements` <collection> unspecified

Return Values

<Bag> new

Errors

If any of the elements of `newElements` do not meet the *element type* constraints of the result object

5.7.26 Protocol: <ByteArray factory>

Conforms To

<initializableCollection factory>

Description

This protocol defines the behavior of objects that can be used to create objects that conform to <ByteArray>. These objects are created with a specified size. If the element values are not explicitly provided, they default to 0.

Standard Globals

`ByteArray`	Conforms to the protocol <ByteArray factory>. Its language element type is unspecified. This is a factory and discriminator for collections that conform to <ByteArray>.

Messages

new
new
with:
with:with:
with:with:with
with:with:with:with:
withAll:

5.7.26.1 Message Refinement: new

Synopsis

Create a new object.

Definition: <instantiator>

Return a newly created object initialized to a standard initial state.

Refinement: <collection factory>

This message has the same effect as sending the message `#new:` with the argument `0`, and will return an empty collection.

Refinement: <ByteArray factory>

Create a new <ByteArray> that contains no elements.

Return Values

<ByteArray> new

Errors

none

5.7.26.2 Message Refinement: new: count

Synopsis

Create a new collection. The parameter `count` constrains the number of *elements* in the result.

Definition: <collection factory>

Return a new collection that has space for at least `count` *elements*.

Conforming protocols may refine this message. In particular, the effect of the parameter `count` should be specified in refinements. It can be used to specify the exact number of *elements*, the minimum number, or in some cases can even be interpreted as a hint from the programmer, with no guarantee that the requested number of instance variables will actually be allocated.

Unless otherwise stated the initial values of *elements* of the new instance of the receiver are unspecified.

Refinement: <ByteArray factory>

The parameter `count` specifies the size of the receiver. The initial value of each *element* of the new instance of the receiver is *0*. The new collections conforms to the protocol <ByteArray>.

Parameters

`count` <integer> unspecified

Return Values

<ByteArray> new

Errors

`count<0`

5.7.26.3 Message Refinement: with: element1
Message Refinement: with: element1 with: element2
Message Refinement: with: element1 with: element2 with: element3
Message Refinement: with: element1 with: element2 with: element3 with: element4

Synopsis

Create a collection initially containing the argument elements

Definition: <initializableCollection factory>

Return a new collection containing a number of elements equal to the number of arguments to this message. The collection contains the arguments as its *elements*.

Conforming protocols may impose restrictions on the values of the arguments and hence the *element types*.

Refinement: <ByteArray factory>

The first argument is at index position 1, the second argument is at index position 2, and so on...

Parameters

`element1` <integer> captured

`element2`	<integer>	captured
`element3`	<integer>	captured
`element4`	<integer>	captured

Return Values

<ByteArray>	new

Errors

If any of the arguments do not meet the *element type* constraints of the result object

5.7.26.4 Message Refinement: withAll: newElements

Synopsis

Create a collection containing only the *elements* of `newElements`.

Definition: <initializableCollection factory>

Return a new collection whose *elements* are the *elements* of `newElements`.

Conforming protocols may impose restrictions on the values of `newElements`.

Refinement: <ByteArray factory>

If the elements of `newElements` are ordered then their ordering establishing their index positions in the new collection.

Parameters

`newElements`	<collection>	unspecified

Return Values

<ByteArray>	new

Errors

If any of the elements of `newElements` do not meet the *element type* constraints of the result object

5.7.27 Protocol: <OrderedCollection factory>

Conforms To

<initializableCollection factory>

Description

This protocol defines the behavior of objects that can be used to create fixed sized ordered collections of objects which can be accessed externally using integer *keys*.

Standard Globals

`OrderedCollection`	Conforms to the protocol <OrderedCollection factory>. Its language element type is unspecified. This is a factory and discriminator for collections that conform to <OrderedCollection>.

Messages

new
new:

with:
with:with:
with:with:with
with:with:with:with:
withAll:

5.7.27.1 Message Refinement: new

Synopsis

Create a new object.

Definition: <instantiator>

Return a newly created object initialized to a standard initial state.

Refinement: <collection factory>

This message has the same effect as sending the message `#new:` with the argument `0`, and will return an empty collection.

Refinement: <OrderedCollection factory>

Create a new <OrderedCollection> that is optimized to store an implementation defined number of *elements*. The new collection initially contains no elements.

Return Values

<OrderedCollection> new

Errors

none

5.7.27.2 Message Refinement: new: count

Synopsis

Create a new collection. The parameter `count` constrains the number of *elements* in the result.

Definition: <collection factory>

Return a new collection that has space for at least `count` *elements*.

Conforming protocols may refine this message. In particular, the effect of the parameter `count` should be specified in refinements. It can be used to specify the exact number of *elements*, the minimum number, or in some cases can even be interpreted as a hint from the programmer, with no guarantee that the requested number of instance variables will actually be allocated.

Unless otherwise stated the initial values of *elements* of the new instance of the receiver are unspecified.

Refinement: <OrderedCollection factory>

The parameter `count` represents a hint for space allocation. The new collection is to optimized to contain `count` *elements*. The new collection initially contains no elements.

Parameters

`count` <integer> unspecified

Return Values

<OrderedCollection> new

Errors

`count<0`

5.7.27.3 Message Refinement: with: element1
Message Refinement: with: element1 with: element2
Message Refinement: with: element1 with: element2 with: element3
Message Refinement: with: element1 with: element2 with: element3 with: element4

Synopsis

Create a collection initially containing the argument elements

Definition: <initializableCollection factory>

Return a new collection containing a number of elements equal to the number of arguments to this message. The collection contains the arguments as its *elements*.

Conforming protocols may impose restrictions on the values of the arguments and hence the *element types*.

Refinement: <OrderedCollection factory>

The first argument is at index position 1, the second argument is at index position 2, and so on.

Parameters

`element1`	<Object>	captured
`element2`	<Object>	captured
`element3`	<Object>	captured
`element4`	<Object>	captured

Return Values

<OrderedCollection> new

Errors

If any of the arguments do not meet the *element type* constraints of the result object

5.7.27.4 Message Refinement: withAll: newElements

Synopsis

Create a collection containing only the *elements* of `newElements`.

Definition: <initializableCollection factory>

Return a new collection whose *elements* are the *elements* of `newElements`.

Conforming protocols may impose restrictions on the values of `newElements`.

Refinement: <OrderedCollection factory>

If the elements of `newElements` are ordered then their ordering establishing their index positions in the new collection.

Parameters

`newElements`	<collection>	unspecified

Return Values

<OrderedCollection> new

Errors

If any of the elements of `newElements` do not meet the *element type* constraints of the result object

5.7.28 Protocol: <Set factory>

Conforms To

<initializableCollection factory>

Description

This protocol defines the behavior of objects that can be used to create objects that conform to the protocol <Set>.

Standard Globals

`Set`	Conforms to the protocol <Set factory>. Its language element type is unspecified. This is a factory and discriminator for collections that conform to <Set>.

Messages

new
new:
with:
with:with:
with:with:with
with:with:with:with:
withAll:

5.7.28.1 Message Refinement: new

Synopsis

Create a new object.

Definition: <instantiator>

Return a newly created object initialized to a standard initial state.

Refinement: <collection factory>

This message has the same effect as sending the message `#new:` with the argument `0`, and will return an empty collection.

Refinement: <Set factory>

Return a new <Set> that is optimized to store an arbitrary number of *elements*. The new collection initially contains no elements.

Return Values

<Set> new

Errors

none

5.7.28.2 Message Refinement: new: count

Synopsis

Create a new collection. The parameter `count` constrains the number of *elements* in the result.

Definition: <collection factory>

Return a new collection that has space for at least `count` *elements*.

Conforming protocols may refine this message. In particular, the effect of the parameter `count` should be specified in refinements. It can be used to specify the exact number of *elements*, the minimum number, or in some cases can even be interpreted as a hint from the programmer, with no guarantee that the requested number of instance variables will actually be allocated.

Unless otherwise stated the initial values of *elements* of the new instance of the receiver are unspecified.

Refinement: <Set factory>

The parameter `count` represents a hint for space allocation. The new collection is to optimized to contain `count` *elements*. If the value of count is zero the collection should be optimize to hold an arbitrary number of elements. The new collection initially contains no elements.

The new collections conforms to the protocol <Set>.

Parameters

`count`	<integer>	unspecified

Return Values

<Set> new

Errors

none

5.7.28.3 Message Refinement: with: element1
Message Refinement: with: element1 with: element2
Message Refinement: with: element1 with: element2 with: element3
Message Refinement: with: element1 with: element2 with: element3 with: element4

Synopsis

Create a collection initially containing the argument elements

Definition: <initializableCollection factory>

Return a new collection containing a number of elements equal to the number of arguments to this message. The collection contains the arguments as its *elements*.

Conforming protocols may impose restrictions on the values of the arguments and hence the *element types*.

Refinement: <Set factory>

The result is undefined if any of the arguments are *nil*.

Parameters

`element1`	<Object>	captured
`element2`	<Object>	captured
`element3`	<Object>	captured
`element4`	<Object>	captured

Return Values

<Set> new

Errors

If any of the arguments do not meet the *element type* constraints of the result object

5.7.28.4 Message Refinement: withAll: newElements

Synopsis

Create a collection containing only the *elements* of `newElements`.

Definition: <initializableCollection factory>

Return a new collection whose *elements* are the *elements* of `newElements`.

Conforming protocols may impose restrictions on the values of `newElements`.

Refinement: <Set factory>

The result is unspecified if `newElements` contains *nil*.

Parameters

`newElements`	<collection>	unspecified

Return Values

<Set> new

Errors

If any of the elements of `newElements` do not meet the *element type* constraints of the result object

5.7.29 Protocol: <SortedCollection factory>

Conforms To

<initializableCollection factory>

Description

Represents protocol for creating a variable sized collection of objects whose *elements* are ordered based on a sort order specified by a two parameter block called the *sort block*. *Elements* may be added, removed or inserted, and can be accessed using external integer *keys*.

Standard Globals

`SortedCollection`	Conforms to the protocol <SortedCollection factory>. Its language element type is unspecified. This is a factory and discriminator for collections that conform to <SortedCollection>.

Messages

new
new:
sortBlock:
with:
with:with:
with:with:with
with:with:with:with:
withAll:

5.7.29.1 Message Refinement: new

Synopsis

Create a new object.

Definition: <instantiator>

Return a newly created object initialized to a standard initial state.

Refinement: <collection factory>

This message has the same effect as sending the message `#new:` with the argument `0`, and will return an empty collection.

Refinement: <SortedCollection factory>

A sort block is supplied which guarantees that the elements will be sorted in ascending order as specified by the #< message for the *elements*. The collection's representation should be optimized to store an arbitrary number of *elements.*

Return Values

SortedCollection new

Errors

none

5.7.29.2 Message Refinement: new: count

Synopsis

Create a new collection. The parameter `count` constrains the number of *elements* in the result.

Definition: <collection factory>

Return a new collection that has space for at least `count` *elements*.

Conforming protocols may refine this message. In particular, the effect of the parameter `count` should be specified in refinements. It can be used to specify the exact number of *elements*, the minimum number, or in some cases can even be interpreted as a hint from the programmer, with no guarantee that the requested number of instance variables will actually be allocated.

Unless otherwise stated the initial values of *elements* of the new instance of the receiver are unspecified.

Refinement: <SortedCollection factory>

The parameter `count` represents an estimate of the maximum number of *elements* in the collection. The representation may be optimized for this size.

A sort block is supplied which guarantees that the elements will be sorted in ascending order as specified by the #< message for the *elements*.

Parameters

`count` <integer> unspecified

Return Values

<SortedCollection> new

Errors

none

5.7.29.3 Message: sortBlock: sortBlock

Synopsis

Create a new sorted collection with `sortBlock` as the sort block.

Definition: <SortedCollection factory>

Return a new sorted collection with `sortBlock` as the sort block. The `sortBlock` specifies the ordering criteria for the new collection and is a two-parameter valuable, which when evaluated with any two *elements* in the receiver, answers *true* if the first parameter should be ordered before the second parameter, and *false* otherwise. The sort block must obey the following properties:

1. Given the same two parameters, the sort block must answer the same result.

2. The sort block must obey transitivity. For example, if a is before b, and b is before c, then a must be before c.

Parameters

`sortBlock` <dyadicValuable> captured

Return Values

<SortedCollection> new

Errors

none

5.7.29.4 Message Refinement: with: element1
Message Refinement: with: element1 with: element2
Message Refinement: with: element1 with: element2 with: element3
Message Refinement: with: element1 with: element2 with: element3 with: element4

Synopsis

Create a collection initially containing the argument elements

Definition: <initializableCollection factory>

Return a new collection containing a number of elements equal to the number of arguments to this message. The collection contains the arguments as its *elements*.

Conforming protocols may impose restrictions on the values of the arguments and hence the *element types*.

Refinement: <SortedCollection factory>

A sort block is supplied which guarantees that the *elements* will be sorted in ascending order as specified by the #< message for the *elements*. The initial *elements* are ordered according to this sort block.

Parameters

`firstElement`	<Object>	captured
`secondElement`	<Object>	captured
`thirdElement`	<Object>	captured
`fourthElement`	<Object>	captured

Return Values

<SortedCollection> new

Errors

If any of the arguments are not appropriate as parameters to the default sort block.

5.7.29.5 Message Refinement: withAll: newElements

Synopsis

Create a collection containing only the *elements* of `newElements`.

Definition: <initializableCollection factory>

Return a new collection whose *elements* are the *elements* of `newElements`.

Conforming protocols may impose restrictions on the values of `newElements`.

Refinement: <SortedCollection factory>

A sort block is supplied which guarantees that the *elements* will be sorted in ascending order as specified by the #< message for the *elements*. The initial *elements* are ordered according to this sort block.

Parameters

`newElements`	<collection>	unspecified

Return Values

<SortedCollection> new

Errors

If any *element* of `newElements` is not appropriate as a parameter to the default sort block.

5.7.30 Protocol: <String factory>

Conforms To

<initializableCollection factory>

Description

This protocol defines the behavior of objects that can be used to create objects that conform to <String>. These objects are created with a specified size.

Standard Globals

`String`	Conforms to the protocol <String factory>. Its language element type is unspecified. This is a factory and discriminator for collections that conform to <String>.

Messages

new
new:
with:
with:with:
with:with:with
with:with:with:with:
withAll:

5.7.30.1 Message Refinement: new

Synopsis

Create a new object.

Definition: <instantiator>

Return a newly created object initialized to a standard initial state.

Refinement: <collection factory>

This message has the same effect as sending the message `#new:` with the argument `0`, and will return an empty collection.

Refinement: <String factory>

Create a new <String> that contains no elements.

Return Values

<String> new

Errors

none

5.7.30.2 Message Refinement: new: count

Synopsis

Create a new collection. The parameter `count` constrains the number of *elements* in the result.

Definition: <collection factory>

Return a new collection that has space for at least `count` *elements*.

Conforming protocols may refine this message. In particular, the effect of the parameter `count` should be specified in refinements. It can be used to specify the exact number of *elements*, the

minimum number, or in some cases can even be interpreted as a hint from the programmer, with no guarantee that the requested number of instance variables will actually be allocated.

Unless otherwise stated the initial values of *elements* of the new instance of the receiver are unspecified.

Refinement: <String factory>

The parameter `count` specifies the size of the receiver. The initial value of each *element* of the new instance of the receiver is unspecified. The new collections conforms to the protocol <String>.

Parameters

`count`	<Integer>	unspecified

Return Values

<String> new

Errors

`count<0`

5.7.30.3 Message Refinement: with: element1
Message Refinement: with: element1 with: element2
Message Refinement: with: element1 with: element2 with: element3
Message Refinement: with: element1 with: element2 with: element3 with: element4

Synopsis

Create a collection initially containing the argument elements

Definition: <initializableCollection factory>

Return a new collection containing a number of elements equal to the number of arguments to this message. The collection contains the arguments as its *elements*.

Conforming protocols may impose restrictions on the values of the arguments and hence the *element types*.

Refinement: <String factory>

The first argument is at index position 1, the second argument is at index position 2, and so on.

Parameters

`element1`	<Character>	captured
`element2`	<Character>	captured
`element3`	<Character>	captured
`element4`	<Character>	captured

Return Values

<String> new

Errors

If any of the arguments do not meet the *element type* constraints of the result object

5.7.30.4 Message Refinement: withAll: newElements

Synopsis

Create a collection containing only the *elements* of `newElements`.

Definition: <initializableCollection factory>

Return a new collection whose *elements* are the *elements* of `newElements`.

Conforming protocols may impose restrictions on the values of `newElements`.

Refinement: <String factory>

If the elements of `newElements` are ordered then their ordering establishing their index positions in the new collection.

Parameters

`newElements` <collection> unspecified

Return Values

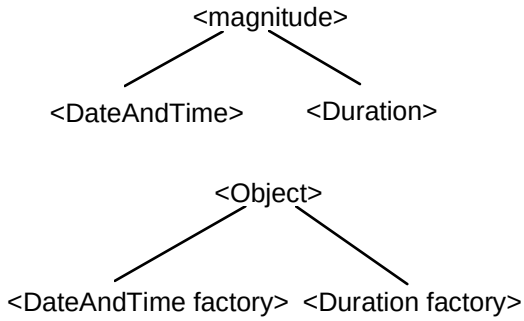
<String> new

Errors

If any of the elements of `newElements` do not meet the *element type* constraints of the result object

5.8 Date and Time Protocols

The standard defines protocols for date and time objects that refer to a specific point in time, and duration objects that represent a length of time.



5.8.1 Protocol: <DateAndTime>

Conforms To

<magnitude>

Description

This protocol describes the behavior that is common to date time objects. Date time objects represent individual points in Coordinated Universal Time (UTC) as represented in an implementation defined local time.

The exact properties of local times are unspecified. Local times may differ in their offset from UTC. A given local time may have different offsets from UTC at different points in time.

All dates and times in the UTC local time are in the Gregorian calendar. Date times prior to the adoption of the Gregorian calendar are given in the retrospective astronomical Gregorian calendar. The year 1 B.C. is astronomical Gregorian year 0. The year 2 B.C. is astronomical Gregorian year -1. The year 1 A.D. is astronomical Gregorian year 1. The offset of the UTC local time is zero.

Messages:

+
-
<
=
>
asLocal
asUTC
dayOfMonth
dayOfWeek
dayOfWeekAbbreviation
dayOfWeekName
dayOfYear
hour
hour12
hour24
isLeapYear
meridianAbbreviation
minute
month
monthAbbreviation
monthName
offset
offset:
printString
second
timeZoneAbbreviation
timeZoneName
year

5.8.1.1 Message: + operand

Synopsis

Answer the result of adding `operand` to the receiver.

Definition: <DateAndTime>

Answer a <DateAndTime> that represents the UTC time that is `operand` after the receiver and whose *local time* is the same as the receiver's. If `operand` is less than <Duration factory> `#zero`, the result is the <DateAndTime> that is that is the absolute value of `operand` before the receiver.

Parameters

`operand` <Duration> uncaptured

Return Values

<DateAndTime> new

Errors

None

5.8.1.2 Message: - operand

Synopsis

Answer the result of adding `operand` to the receiver.

Definition: <DateAndTime>

If `operand` is a <DateAndTime>, answer a <Duration> whose value is the period of time between the `operand` and the receiver. If `operand` is a <DateAndTime> prior to the receiver then the result is a <Duration> less than <Duration factory> `#zero`.

If `operand` is a <Duration>, answer a new <DateAndTime> which represents the UTC time that is `operand` before the receiver and whose *local time* is the same as the receiver's. If `operand` is a duration less than <Duration factory> `#zero` then the result is a <DateAndTime> that is the absolute value of operand after the receiver.

Parameters

`operand` <DateAndTime> | <Duration> uncaptured

Return Values

<Duration> unspecified

<DateAndTime> unspecified

Errors

none.

5.8.1.3 Message Refinement: < operand

Synopsis

Answer *true* if the receiver is less than operand. Answer *false* otherwise.

Definition: <magnitude>

Answer *true* if the receiver is less than operand with respect to the ordering defined for them. Answer *false* otherwise.

It is erroneous if the receiver and operand are not *comparable*.

The semantics of the natural ordering must be defined by refinement, which may also restrict the type of operand.

Refinement: <DateAndTime>

Answer true if the UTC time represented by `operand` follows the UTC time represented by the receiver. Answer false otherwise.

If the offsets of the receiver and operand are the same then their order is determined by their *lexical order* in the sequence `#year`, `#month`, `#day`, `#hour24`, `#minute`, `#second`. If their offsets differ then result is the same as if `receiver asUTC < operand asUTC` were evaluated.

Parameters

`operand` <DateAndTime> uncaptured

Return Values

<boolean> unspecified

Errors

none

5.8.1.4 Message Refinement: = comparand

Synopsis

Object equivalence test.

Definition: <Object>

This message tests whether the receiver and the `comparand` are equivalent objects at the time the message is processed. Return *true* if the receiver is equivalent to `comparand`. Otherwise return *false*.

The meaning of "equivalent" cannot be precisely defined but the intent is that two objects are considered equivalent if they can be used interchangeably. Conforming protocols may choose to more precisely define the meaning of "equivalent".

The value of

```
receiver = comparand
```

is *true* if and only if the value of

```
comparand = receiver
```

would also be *true*. If the value of

```
receiver = comparand
```

is *true* then the receiver and `comparand` must have *equivalent hash values*. Or more formally:

```
receiver = comparand
receiver hash = comparand hash
```

The equivalence of objects need not be *temporally invariant*. Two independent invocations of #= with the same receiver and `operand` objects may not always yield the same results. Note that a collection that uses #= to discriminate objects may only reliably store objects whose hash values do not change while the objects are contained in the collection.

Refinement: <DateAndTime>

Answer true if the `comparand conforms to <DateAndTime> and if it` represents the same UTC time as the receiver. Answer false otherwise. The *local times* of the receiver and operand are ignored.

Parameters

`comparand`	<Object>	uncaptured

Return Values

<boolean>	unspecified

Errors

none

5.8.1.5 Message Refinement: > operand

Synopsis

Answer *true* if the receiver is greater than operand. Answer *false* otherwise.

Definition: <magnitude>

Answer *true* if the receiver is greater than operand with respect to the natural ordering. Answer *false* otherwise.

It is erroneous if the receiver and operand are not *comparable*.

The semantics of the natural ordering must be defined by refinement, which may also restrict the type of operand.

Refinement: <DateAndTime>

Answer true if the UTC time represented by `operand` precedes the UTC time represented by the receiver. Answer false otherwise.

If the offsets of the receiver and operand are the same then their order is determined by their *lexical order* in the sequence `#year`, `#month`, `#day`, `#hour24`, `#minute`, `#second`. If their offsets differ then result is the same as if `receiver asUTC > operand asUTC` were evaluated.

Parameters

`operand` <DateAndTime> uncaptured

Return Values

<boolean> unspecified

Errors

none

5.8.1.6 Message: asLocal

Synopsis

Answer a <DateAndTime> that represents the same UTC time as the receiver but in the *local time* specified by the implementation.

Definition: <DateAndTime>

Answer a <DateAndTime> that represents the same UTC time as the receiver but in the *local time* specified by the implementation.

Return Values

<DateAndTime> unspecified

Errors

None

5.8.1.7 Message: asUTC

Synopsis

Answer a <DateAndTime> that represents the same absolute time as the receiver but in the *local time* UTC.

Definition: <DateAndTime>

Answer a <DateAndTime> that represents the same absolute time as the receiver but in the *local time* UTC. The exact meaning of UTC *local time* is specified by the implementation. The UTC *local time* must use the Gregorian calendar. <DateAndTimes> representing UTC times prior to the adoption of the Gregorian calendar must use the retrospective astronomical Gregorian calendar. It is an invariant that

```
<DateAndTime> asUTC offset = Duration zero.
```

Return Values

<DateAndTime> unspecified

Errors

None

5.8.1.8 Message: dayOfMonth

Synopsis

Answer the number of the day in the month in the *local time* of the receiver which includes the receiver.

Definition: <DateAndTime>

Answer an <integer> between 1 and 31 inclusive representing the number of the day in the month, in the *local time* of the receiver, which includes the receiver.

Return Values

<integer> unspecified

Errors

None

5.8.1.9 Message: dayOfWeek

Synopsis

Answer the number of the day in the week, in the *local time* of the receiver, which includes the receiver.

Definition: <DateAndTime>

Answer an <integer> between 1 and 7 inclusive representing the number of the day in the week, in the *local time* of the receiver, which includes the receiver. Sunday is 1, Monday is 2, and so on.

Return Values

<integer> unspecified

Errors

None

5.8.1.10 Message: dayOfWeekAbbreviation

Synopsis

Answer the abbreviation of the name, in the *local time* of the receiver, of the day of the week which includes the receiver.

Definition: <DateAndTime>

Answer an <readableString> which is the abbreviation of the name, in the *local time* of the receiver, of the day of the week which includes the receiver.

Return Values

<readableString> unspecified

Errors

None

5.8.1.11 Message: dayOfWeekName

Synopsis

Answer the name, in the *local time* of the receiver, of the day of the week which includes the receiver.

Definition: <DateAndTime>

Answer an <readableString> which is the name, in the *local time* of the receiver, of the day of the week which includes the receiver.

Return Values

<readableString> unspecified

Errors

None

5.8.1.12 Message: dayOfYear

Synopsis

Answer the number of the day in the year, in the *local time* of the receiver, which includes the receiver.

Definition: <DateAndTime>

Answer an <integer> between 1 and 366 inclusive representing the number of the day in the year, in the *local time* of the receiver, which includes the receiver.

Return Values

<integer> unspecified

Errors

None

5.8.1.13 Message: hour

Synopsis

Answer the number of the hour in the day, in the *local time* of the receiver, which includes the receiver.

Definition: <DateAndTime>

Answer an <integer> between 0 and 23 inclusive representing the number of the hour in the day, in the *local time* of the receiver, which includes the receiver. It is implementation defined whether a given *local time* uses the 12-hour clock or the 24-hour clock, except that the UTC *local time* must use the 24-hour clock.

Return Values

<integer> unspecified

Errors

None

5.8.1.14 Message: hour12

Synopsis

Answer the hour in the day in the 12-hour clock of the *local time* of the receiver.

Definition: <DateAndTime>

Answer an <integer> between 1 and 12 inclusive representing the hour in the day in the 12-hour clock of the *local time* of the receiver.

Return Values

<integer> unspecified

Errors

None

5.8.1.15 Message: hour24

Synopsis

Answer the hour in the day in the 24-hour clock of the *local time* of the receiver.

Definition: <DateAndTime>

Answer an <integer> between 0 and 23 inclusive representing the hour in the day in the 24-hour clock of the *local time* of the receiver.

Return Values

<integer> unspecified

Errors

None

5.8.1.16 Message: isLeapYear

Synopsis

Test for leap year.

Definition: <DateAndTime>

Answer true if the year, which includes the receiver, in the *local time* of the receiver is a leap year, false otherwise.

Two <DateAndTime> objects that are equal can give different results for #isLeapYear. Equality depends on their UTC time whereas #isLeapYear depends on their *local time*.

Return Values

<boolean> unspecified

Errors

None

5.8.1.17 Message: meridianAbbreviation

Synopsis

Answer the abbreviation, in the *local time* of the receiver, of the name of the half of the day, which includes the receiver.

Definition: <DateAndTime>

Answer a <readableString> that is the abbreviation, in the *local time* of the receiver, of the name of the half of the day, which includes the receiver.

Return Values

<readableString> unspecified

Errors

None

5.8.1.18 Message: minute

Synopsis

Answer the minute of the hour in the *local time* of the receiver.

Definition: <DateAndTime>

Answer an <integer> between 0 and 59 inclusive representing the minute of hour in the *local time* of the receiver.

Return Values

<integer>unspecified

Errors

None

5.8.1.19 Message: month

Synopsis

Answer the number of the month in the year, in the *local time* of the receiver, which includes the receiver.

Definition: <DateAndTime>

Answer an <integer> between 1 and 12 inclusive representing the number of the month in the year, in the *local time* of the receiver, which includes the receiver.

Return Values

<integer>unspecified

Errors

None

5.8.1.20 Message: monthAbbreviation

Synopsis

Answer the abbreviation of the name of the month, in the *local time* of the receiver, which includes the receiver.

Definition: <DateAndTime>

Answer a <readableString> that is the abbreviation of the name of the month, in the *local time* of the receiver, which includes the receiver.

Return Values

<readableString> unspecified

Errors

None

5.8.1.21 Message: monthName

Synopsis

Answer the name of the month, in the *local time* of the receiver, which includes the receiver.

Definition: <DateAndTime>

Answer a <readableString> that is the name of the month, in the *local time* of the receiver, which includes the receiver.

Return Values

<readableString> unspecified

Errors

None

5.8.1.22 Message: offset

Synopsis

Answer the difference between the *local time* of the receiver and UTC at the time of the receiver.

Definition: <DateAndTime>

Answer a <Duration> representing the difference between the *local time* of the receiver and UTC at the time of the receiver.

Return Values

<Duration> unspecified

Errors

None

5.8.1.23 Message: offset: offset

Synopsis

Answer a <DateAndTime> equivalent to the receiver but with its *local time* being offset from UTC by `offset`.

Definition: <DateAndTime>

Answer a <DateAndTime> equivalent to the receiver but with its *local time* being offset from UTC by `offset`. The impact of this on any other *local time* property is unspecified.

Implementations may define a limit to the range of `offset`, but it must be at least -12:00:00 to 12:00:00 inclusive.

It is an invariant that if x is a <Duration> in range then
`(<DateAndTime> offset: x) offset = x`

Parameters

offset <Duration> unspecified

Return Values

<DateAndTime> unspecified

Errors

`offset` out of range

5.8.1.24 Message Refinement: printString

Synopsis

Return a string that describes the receiver.

Definition: <Object>

A string consisting of a sequence of characters that describe the receiver are returned as the result.

The exact sequence of characters that describe an object are implementation defined.

Refinement: <DateAndTime>

The returned string will represent the UTC time of the receiver offset from UTC by the offset of the receiver. All dates are in the astronomical Gregorian calendar. The result will be formatted as

-YYYY-MM-DDThh:mm:ss.s+ZZ:zz:z where

-	is the <Character> $- if the year is less than 0 otherwise it is the <Character> that is returned from the message `#space` sent to the standard global `Character`,
YYYY	is the year left zero filled to four places,
-	is the <Character> $-,
MM	is the month of the year left zero filled to two places,
-	is the <Character> $-,
DD	is the day of the month left zero filled to two places,
T	is the <Character> $T,
hh	is the hour in the 24-hour clock left zero filled to two places,
:	is the <Character> $:,
mm	is the minute left zero filled to two places,
:	is the <Character> $:,
ss	is the second left zero filled to two places,
.	is the <Character> $. and is present only if the fraction of a second is non-zero,
s	is the fraction of a second and is present only if non-zero,
+	is the <Character> $+ if the offset is greater than or equal to <Duration factory> #zero and the <Character> $- if it is less,
ZZ	is the hours of the offset left zero filled to two places, and
:	is the <Character> $:,
zz	is the minutes of the offset left zero filled to two places,
:	is the <Character> $: and is present only if the seconds of the offset is non-zero,
z	is the seconds of the offset including any fractional part and is present only if non-zero.

This format is based on ISO 8601 sections 5.3.3 and 5.4.1.

Example: 8:33:14.321 PM EST January 5, 1200 B.C.

```
'-1199-01-05T20:33:14.321-05:00'
```

Example: 12 midnight UTC January 1, 2001 A.D.

```
' 2001-01-01T00:00:00+00:00'
```

Return Values

<readableString> unspecified

Errors

none

5.8.1.25 Message: second

Synopsis

Answer the second of the minute of the *local time* of the receiver.

Definition: <DateAndTime>

Answer a <number> greater than or equal to 0 and strictly less than 60 representing the second of the minute of the *local time* of the receiver.

Return Values

<number> unspecified

Errors

None

5.8.1.26 Message: timeZoneAbbreviation

Synopsis

Answer the abbreviation of the name, in the *local time* of the receiver, of the time zone of the receiver.

Definition: <DateAndTime>

Answer a <readableString> that is the abbreviation of the name, in the *local time* of the receiver, of the time zone of the receiver.

Return Values

<readableString> unspecified

Errors

None

5.8.1.27 Message: timeZoneName

Synopsis

Answer the name in the *local time* of the receiver of the time zone of the receiver.

Definition: <DateAndTime>

Answer a <readableString> that is the name in the *local time* of the receiver of the time zone of the receiver.

Return Values

<readableString> unspecified

Errors

None

5.8.1.28 Message: year

Synopsis

Answer the number of the year in the *local time* of the receiver which includes the receiver.

Definition: <DateAndTime>

Answer an<integer> the number of the year which includes the receiver.

Return Values

<integer>unspecified

Errors

None

5.8.2 Protocol: <Duration>

Conforms To

<magnitude>

Description

Represents a length of time.

Messages:

*
+
-
/
<
=
>
asSeconds
abs
days
hours
minutes
negated
negative
positive
printString
seconds

5.8.2.1 Message: * operand

Synopsis

Answer the result of multiplying the receiver by `operand`.

Definition: <Duration>

Answer a <Duration> that is the result of multiplying the receiver by `operand`.

Parameters

`operand` <number> unspecified

Return Values

<Duration> new

Errors

None

5.8.2.2 Message: + operand

Synopsis

Answer the result of adding `operand` to the receiver.

Definition: <Duration>

Answer a <Duration> whose value is the result of adding the receiver and `operand`.

Parameters

`operand`	<Duration>	unspecified

Return Values

<Duration> new

Errors

None

5.8.2.3 Message: - operand

Synopsis

Answer the result of subtracting the operand from the receiver.

Definition: <Duration>

Answer a <Duration> whose value is the result of subtracting `operand` from the receiver.

Parameters

`operand`	<Duration>	unspecified

Return Values

<Duration>	new

Errors

None

5.8.2.4 Message: / operand

Synopsis

Answer the result of dividing the receiver by `operand`.

Definition: <Duration>

If `operand` is a <number> answer a new <Duration> whose value is the result of dividing the receiver by `operand`. If `operand` equals zero the ZeroDivide exception is signaled.

If `operand` is a <Duration> answer a <number> whose value is the result of dividing the receiver by `operand`. If `operand` is `<Duration factory> #zero` the ZeroDivide exception is signaled.

Parameters

`operand`	<number> | <Duration>	unspecified

Return Values

<Duration>	unspecified
<number>	unspecified

Errors

none

5.8.2.5 Message Refinement: < operand

Synopsis

Answer *true* if the receiver is less than operand. Answer *false* otherwise.

Definition: <magnitude>

Answer *true* if the receiver is less than operand with respect to the ordering defined for them. Answer *false* otherwise.

It is erroneous if the receiver and operand are not *comparable*.

The semantics of the natural ordering must be defined by refinement, which may also restrict the type of operand.

Refinement: <Duration>

Answer true if operand represents a <Duration> that is larger than the receiver. Answer false otherwise.

Parameters

operand <Duration> unspecified

Return Values

<boolean> unspecified

Errors

none

5.8.2.6 Message Refinement: = comparand

Synopsis

Object equivalence test.

Definition: <Object>

This message tests whether the receiver and the `comparand` are equivalent objects at the time the message is processed. Return *true* if the receiver is equivalent to `comparand`. Otherwise return *false*.

The meaning of "equivalent" cannot be precisely defined but the intent is that two objects are considered equivalent if they can be used interchangeably. Conforming protocols may choose to more precisely define the meaning of "equivalent".

The value of

```
receiver = comparand
```

is *true* if and only if the value of

```
comparand = receiver
```

would also be *true*. If the value of

```
receiver = comparand
```

is *true* then the receiver and `comparand` must have *equivalent hash values*. Or more formally:

```
receiver = comparand
receiver hash = comparand hash
```

The equivalence of objects need not be *temporally invariant*. Two independent invocations of #= with the same receiver and `operand` objects may not always yield the same results. Note that a collection that uses #= to discriminate objects may only reliably store objects whose hash values do not change while the objects are contained in the collection.

Refinement: <Duration>

Answer true if the `comparand` is a <Duration> representing the same length of time as the receiver. Answer false otherwise.

Parameters

`comparand` <Object> uncaptured

Return Values

<boolean> unspecified

Errors

none

5.8.2.7 Message Refinement: > operand

Synopsis

Answer *true* if the receiver is greater than operand. Answer *false* otherwise.

Definition: <magnitude>

Answer *true* if the receiver is greater than operand with respect to the natural ordering. Answer *false* otherwise.

It is erroneous if the receiver and operand are not *comparable*.

The semantics of the natural ordering must be defined by refinement, which may also restrict the type of operand.

Refinement: <Duration>

Answer true if `operand` represents a <Duration> which is smaller than the receiver. Answer false otherwise.

Parameters

`operand` <Duration> unspecified

Return Values

<boolean> unspecified

Errors

none

5.8.2.8 Message: asSeconds

Synopsis

Answer the total number of seconds in the length of time represented by the receiver.

Definition: <Duration>

Answer the total number of seconds in the length of time represented by the receiver including any fractional part of a second. If the receiver is less than <Duration factory> `#zero` then the result will be less than 0.

Return Values

<number> unspecified

Errors

None

5.8.2.9 Message: abs

Synopsis

Answer the absolute value of the receiver.

Definition: <Duration>

If the receiver is greater than or equal to <Duration Factory> `#zero` answer a <Duration> which is equal to the receiver. Otherwise answer a <Duration> which has the same magnitude as the receiver but the opposite sign.

Return Values

<Duration> unspecified

Errors

None

5.8.2.10 Message: days

Synopsis

Answer the number of complete days in the receiver.

Definition: <Duration>

Answer the number of complete days in the receiver. If the receiver is less than <Duration factory> `#zero` then the result will be less than or equal to 0.

Return Values

<integer> unspecified

Errors

None

5.8.2.11 Message: hours

Synopsis

Answer the number of complete hours in the receiver.

Definition: <Duration>

Answer an <integer> between -23 and 23 inclusive that represents the number of complete hours in the receiver, after the number of complete days has been removed. If the receiver is less than <Duration factory> `#zero` then the result will be less than or equal to 0.

Return Values

<integer>unspecified

Errors

None

5.8.2.12 Message: minutes

Synopsis

Answer the number of complete minutes in the receiver.

Definition: <time>

Answer an <integer> between -59 and 59 inclusive that represents the number of complete minutes in the receiver, after the number of complete days and hours have been removed. If the receiver is less than <Duration factory> `#zero` then the result will be less than or equal to 0.

Return Values

<integer>unspecified

Errors

None

5.8.2.13 Message: negated

Synopsis

Answer the negation of the receiver.

Definition: <Duration>

Answer a <Duration> which is of the same magnitude but opposite sign as the receiver.

Return Values

<Duration> unspecified

Errors

None

5.8.2.14 Message: negative

Synopsis

Answer true if the receiver is less than <Duration factory> `#zero`.

Definition: <Duration>

Answer true if the receiver is less than <Duration factory> `#zero`, false otherwise.

Return Values

<boolean> unspecified

Errors

None

5.8.2.15 Message: positive

Synopsis

Answer true if the receiver is greater than or equal to <Duration factory> `#zero`.

Definition: <Duration>

Answer true if the receiver is greater than or equal to the <Duration factory> `#zero`, false otherwise.

Return Values

<boolean> unspecified

Errors

None

5.8.2.16 Message Refinement: printString

Synopsis

Return a string that describes the receiver.

Definition: <Object>

A string consisting of a sequence of characters that describe the receiver are returned as the result.

The exact sequence of characters that describe an object is implementation defined.

Refinement: <Duration>

Answer a description of the receiver that is formatted as

[-]D:HH:MM:SS[.S] where

-	is a minus sign if the receiver represents a length of time going from the future into the past,
D	is the number of complete days with leading zeros to fill one place,
HH	is the number of complete hours with leading zeros to fill two places,
MM	is the number of complete minutes with leading zeros to fill two places,
SS	is. the number of complete seconds with leading zeros to fill two places, and
.S	is the fractional part of the number of seconds, if any.

Return Values

<readableString> unspecified

Errors

None

5.8.2.17 Message: seconds

Synopsis

Answer the number of seconds in the receiver.

Definition: <Duration>

Answer a <number> strictly greater than -60 and strictly less than 60 that represents the number of seconds in the receiver, after the complete days, hours, and minutes have been removed. If the receiver is less than <Duration factory> `#zero` then the result will be less than or equal to 0.

Return Values

<number> unspecified

Errors

None

5.8.3 Protocol: <Duration factory>

Conforms To

<Object>

Description

Represents protocol for creating a particular length of time.

Standard Globals

`Duration`	Conforms to the protocol <Duration factory>. Its language element type is unspecified.

Messages:

days:hours:minutes:seconds:
seconds:
zero

5.8.3.1 Message: days: days hours: hours minutes: minutes seconds: seconds

Synopsis

Answer a <Duration> of the number of days, hours, minutes, and seconds.

Definition: <Duration factory>

Answer a <Duration> of the number of days, hours, minutes, and seconds. If any of the operands are negative, the result is smaller by that number of days, hours, minutes, or seconds as appropriate.

Parameters

`days`	<integer>	unspecified
`hours`	<integer>	unspecified

`minutes`	<integer>	unspecified
`seconds`	<number>	unspecified

Return Values

<Duration> new

5.8.3.2 Message: seconds: seconds

Synopsis

Answer a <Duration> which is `seconds` in length

Definition: <Duration factory>

If `seconds` is negative, answer a <Duration> that is abs (`seconds`) less than <Duration factory> `#zero`.

Parameters

seconds	<number>	unspecified

Return Values

<Duration>	new

5.8.3.3 Message: zero

Synopsis

Answer a <Duration> of zero length.

Definition: <Duration factory>

Answer a <Duration> representing a length of no time.

Return Values

<Duration> unspecified

5.8.4 Protocol: <DateAndTime factory>

Conforms To

<Object>

Description

Represents protocol for creating an abstraction for a particular day of the year.

Standard Globals

`DateTime`	Conforms to the protocol <DateAndTime factory>. Its language element type is unspecified.

Messages:

clockPrecision
year:month:day:hour:minute:second:
year:month:day:hour:minute:second:offset:
year:day:hour:minute:second:
year:day:hour:minute:second:offset:
now

5.8.4.1 Message: clockPrecision

Synopsis

Answer a <Duration> such that after that period of time passes, `#now` is guaranteed to give a different result.

Definition: <DateAndTime factory>

Answer a <Duration> such that after that period of time passes, `#now` is guaranteed to give a different result. Ideally implementations should answer the least such duration.

Return Values:

<Duration> unspecified

Errors

None

5.8.4.2 Message: year: year month: month day: dayOfMonth hour: hour minute: minute second: second

Synopsis

Answer a <DateAndTime> which is the second `second` of the minute `minute` of the hour `hour` of the day `dayOfMonth` of the month `month` of the year `year` of the astronomical Gregorian calendar in local time.

Definition: <DateAndTime factory>

Answer the least <DateAndTime> which is the second `second` of the minute `minute` of the hour `hour` of the day `dayOfMonth` of the month `month` of the year `year` of the astronomical Gregorian calendar in the *local time* specified by the implementation. The `second` must be a <number> greater than or equal to 0 and strictly less than 60. The `minute` must be an <integer> between 0 and 59 inclusive. The `hour` must be an <integer> between 0 and 23 inclusive. The `day` must be an <integer> between 1 and 31 inclusive. The `month` must be an <integer> between 1 and 12 inclusive. An implementation may not impose any limits on the year other than those imposed on <integer> constants.

It is possible that the time specified does not exist in the local time defined by the implementation. If there is a time change such that the *local time* is set forward and the time specified is in the interregnum, then that time does not exist in the local time. For example if at 02:00 in California on April 26, 1997 there is a time change that sets local time forward one hour, then the local time 02:30 in California does not exist. Conversely if there is a time change that sets the *locale time* back there are times which are ambiguous. For example if instead of setting the local time forward from 02:00 to 03:00 it is set back to 01:00 the the local time 01:30 in California is ambiguious. The result is the least <DateAndTime> that conforms to the given parameters.

Parameters

`year`	<integer>	unspecified
`month`	<integer>	unspecified
`dayOfMonth`	<integer>	unspecified
`hour`	<integer>	unspecified
`minute`	<integer>	unspecified
`second`	<number>	unspecified

Return Values

<DateAndTime> new

Errors

`month` is not between 1 and 12 inclusive.

`dayOfMonth` greater than the number of days in the month `month` of year `year` of the astronomical Gregorian calendar.

`hour` is not between 0 and 23 inclusive.

`minute` is not between 0 and 59 inclusive.

`second` is not greater than or equal to 0 and strictly less than 60.

the time specified does not exist.

5.8.4.3 Message: year: year month: month day: dayOfMonth hour: hour minute: minute second: second offset: offset

Synopsis

Answer a <DateAndTime> which is the second `second` of the minute `minute` of the hour `hour` of the day `dayOfMonth` of the month `month` of the year `year` of the astronomical Gregorian calendar offset from UTC by `offset`.

Definition: <DateAndTime factory>

Answer the least <DateAndTime> which is the second `second` of the minute `minute` of the hour `hour` of the day `dayOfMonth` of the month `month` of the year `year` of the astronomical Gregorian calendar offset from UTC by `offset`. The `second` must be a <number> greater than or equal to 0 and strictly less than 60. The `minute` must be an <integer> between 0 and 59 inclusive. The `hour` must be an <integer> between 0 and 23 inclusive. The `day` must be an <integer> between 1 and 31 inclusive. The `month` must be an <integer> between 1 and 12 inclusive. An implementation may not impose any limits on the year other than those imposed on <integer> constants.

It is possible that the time specified does not exist in the local time defined by the implementation. If there is a time change such that the *local time* is set forward and the time specified is in the interregnum, then that time does not exist in the local time. For example if at 02:00 in California on April 26, 1997 there is a time change that sets local time forward one hour, then the local time 02:30 in California does not exist. Conversely if there is a time change that sets the *locale time* back there are times which are ambiguous. For example if instead of setting the local time forward from 02:00 to 03:00 it is set back to 01:00 the the local time 01:30 in California is ambiguious. The result is the least <DateAndTime> that conforms to the given parameters.

Parameters

`year`	<integer>	unspecified
`month`	<integer>	unspecified
`dayOfMonth`	<integer>	unspecified
`hour`	<integer>	unspecified
`minute`	<integer>	unspecified
`second`	<number>	unspecified
`offset`	<Duration>	unspecified

Return Values

<DateAndTime> new

Errors

`month` is not between 1 and 12 inclusive.

`dayOfMonth` greater than the number of days in the month `month` of year `year` of the astronomical Gregorian calendar.

`hour` is not between 0 and 23 inclusive.

`minute` is not between 0 and 59 inclusive.

`second` is not greater than or equal to 0 and strictly less than 60.

5.8.4.4 Message: year: year day: dayOfYear hour: hour minute: minute second: second

Synopsis

Answer a <DateAndTime> which is the second `second` of the minute `minute` of the hour `hour` of the day `dayOfYear` of the year `year` of the astronomical Gregorian calendar in local time.

Definition: <DateAndTime factory>

Answer the least <DateAndTime> which is the second `second` of the minute `minute` of the hour `hour` of the day `dayOfYear` of the year `year` of the astronomical Gregorian calendar in the local time specified by the implementation. The `second` must be a <number> greater than or equal to 0 and strictly less than 60. The `minute` must be an <integer> between 0 and 59 inclusive. The `hour` must be an <integer> between 0 and 23 inclusive. The `day` must be an <integer> between 1 and 366 inclusive. An implementation may not impose any limits on the year other than those imposed on <integer> constants.

It is possible that the time specified does not exist in the local time specified by the implementation. If there is a time change such that the local time is set forward and the time specified is in the interregnum, then that time does not exist in the local time. For example if at 02:00 in California on April 26, 1997 there is a time change that sets local time forward one hour, then the local time 02:30 in California does not exist. Conversely if there is a time change that sets the *locale time* back there are times which are ambiguous. For example if instead of setting the local time forward from 02:00 to 03:00 it is set back to 01:00 the the local time 01:30 in California is ambiguious. The result is the least <DateAndTime> that conforms to the given parameters.

It is worth noting that the year 1 B.C. is year 0 in the astronomical Gregorian calendar. Similarly the year 2 B.C. is year -1 in the astronomical Gregorian calendar and so on. The year 1 A.D. is year 1 in the astronomical Gregorian calendar.

Parameters

`year`	<integer>	unspecified
`dayOfYear`	<integer>	unspecified
`hour`	<integer>	unspecified
`minute`	<integer>	unspecified
`second`	<number>	unspecified

Return Values

<DateAndTime> new

Errors

`month` is not between 1 and 12 inclusive.

`dayOfYear` greater than the number of days in the year `year` of the astronomical Gregorian calendar.

`hour` is not between 0 and 23 inclusive.

`minute` is not between 0 and 59 inclusive.

`second` is not greater than or equal to 0 and strictly less than 60.

the time specified does not exist.

5.8.4.5 Message: year: year day: dayOfYear hour: hour minute: minute second: second offset: offset

Synopsis

Answer a <DateAndTime> which is the second `second` of the minute `minute` of the hour `hour` of the day `dayOfYear` of the year `year` of the astronomical Gregorian calendar offset from UTC by `offset`.

Definition: <DateAndTime factory>

Answer the least <DateAndTime> which is the second `second` of the minute `minute` of the hour `hour` of the day `dayOfYear` of the year `year` of the astronomical Gregorian calendar in the local time of the locale `locale`. The `second` must be a <number> greater than or equal to 0 and strictly less than 60. The `minute` must be an <integer> between 0 and 59 inclusive. The `hour` must be an <integer> between 0 and 23 inclusive. The `day` must be an <integer> between 1 and 366 inclusive. An implementation may not impose any limits on the year other than those imposed on <integer> constants.

It is possible that the time specified does not exist in the local time defined by the implementation. If there is a time change such that the *local time* is set forward and the time specified is in the interregnum, then that time does not exist in the local time. For example if at 02:00 in California on April 26, 1997 there is a time change that sets local time forward one hour, then the local time 02:30 in California does not exist. Conversely if there is a time change that sets the *locale time* back there are times which are ambiguous. For example if instead of setting the local time forward from 02:00 to 03:00 it is set back to 01:00 the the local time 01:30 in California is ambiguious. The result is the least <DateAndTime> that conforms to the given parameters.

Parameters

`year`	<integer>	unspecified
`dayOfYear`	<integer>	unspecified
`hour`	<integer>	unspecified
`minute`	<integer>	unspecified
`second`	<number>	unspecified
`offset`	<Duration>	unspecified

Return Values

<DateAndTime> new

Errors

`month` is not between 1 and 12 inclusive.

`dayOfYear` greater than the number of days in the year `year` of the astronomical Gregorian calendar.

`hour` is not between 0 and 23 inclusive.

`minute` is not between 0 and 59 inclusive.

`second` is not greater than or equal to 0 and strictly less than the number of seconds in the minute specified.

5.8.4.6 Message: now

Synopsis

Answer a <DateAndTime> representing the current date and time.

Definition: <DateAndTime factory>

Answer a <DateAndTime> representing the current date and time in the *local time* specified by the implementation.

Return Values

<DateAndTime> unspecified

Errors

None

5.9 Stream Protocols

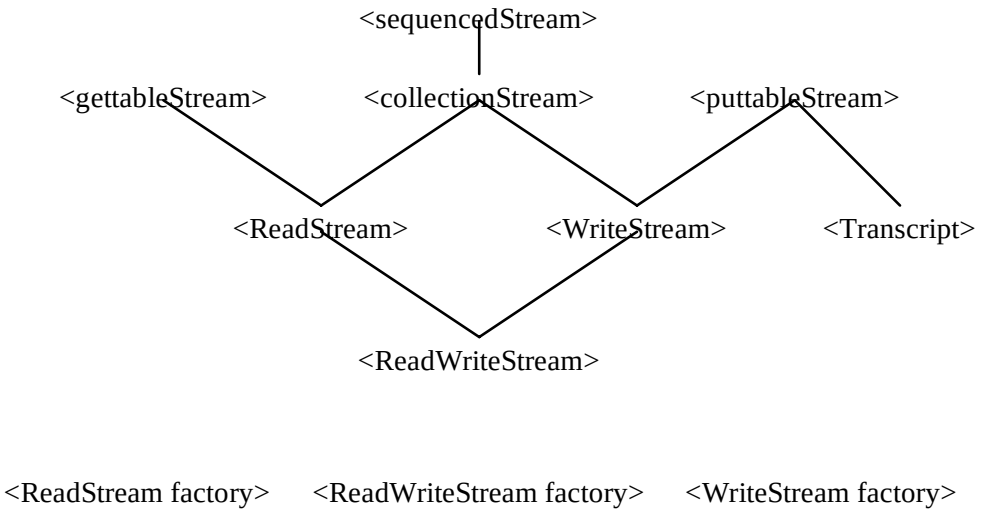
This section includes protocols that define the fundamental behavior of various kinds of streams. Streams produce or consume a sequence of values. Some stream classes will build sequenceable collections or report the values of a sequenceable collection. Other types of streams may operate on files, positive integers, random numbers, and so forth.

There are seven protocols that describe stream behavior. <sequencedStream> describes a stream on a sequence of objects and allows for positioning of the stream.<gettableStream> allows for reading from a stream. <puttableStream> allows for writing to a stream. <collectionStream> provides for the association of a stream with a collection. <ReadStream> reads a sequence of objects from a preexisting collection and can peek at objects prior to reading them. The objects written to a <WriteStream> are accumulated so they can be independently accessed as a collection; <ReadWriteStream> can read, peek, and write within a collection of objects.

The protocol <Transcript> defines the behavior of the object that is the value of the global named Transcript. Transcript is a stream that may be used to log textual message generated by a Smalltalk program.

There are three factory protocols that specify the behavior of three global stream factories, ReadStream, WriteStream, and ReadWriteStream, used to create various types of streams.

The graph below shows the conformance relationships between the protocols defined in this section.



5.9.1 Protocol: <sequencedStream>

Conforms To

<Object>

Description

An object conforming to <sequencedStream> has a finite number of *past* and *future sequence values*. It maintains a position on its *sequence values* and allows the position to be altered.

Messages

close
contents
isEmpty
position
position:
reset
setToEnd

5.9.1.1 Message: close

Synopsis

Disassociate a stream from its backing store.

Definition: <sequencedStream>

If the receiver is a *write-back stream* update its *stream backing store* as if the message #flush was sent to the receiver. Then eliminate any association between the receiver and its stream backing store. Any system resources associated with the association should be released. The effect of sending any message to the receiver subsequent to this message is undefined.

Return Value

UNSPECIFIED

Errors

none

5.9.1.2 Message: contents

Synopsis

Returns a collection containing the complete contents of the stream.

Definition: <sequencedStream>

Returns a collection that contains the receiver's *past* and *future sequence values*, in order. The size of the collection is the sum of the sizes of the *past* and *future sequence values*.

Return Value

<sequencedReadableCollection> unspecified

Errors

none

5.9.1.3 Message: isEmpty

Synopsis

Returns a Boolean indicating whether there are any *sequence values* in the receiver.

Definition: <sequencedStream>

Returns *true* if both the set of *past* and *future sequence values* of the receiver are empty. Otherwise returns *false.*

Return Value

<boolean> unspecified

Errors

none

5.9.1.4 Message: position

Synopsis

Returns the current position of the stream.

Definition: <sequencedStream>

Returns the number of *sequence values* in the receiver's *past sequence values*.

Return Value

<integer>unspecified

Errors

none

5.9.1.5 Message: position: amount

Synopsis

Sets the current position in a stream of values.

Definition: <sequencedStream>

If the number of *sequence values* in the receiver's *past sequence values* is smaller than `amount`, move objects in sequence from the front of the receiver's *future sequence values* to the back of th receiver's *past sequence values* until the number of *sequence values* in the receiver's *past sequence values* is equal to `amount`.

If the number of *sequence values* in the receiver's *past sequence values* is greater than `amount`, move objects in sequence from the back of the receiver's *past sequence values* to the front of th receiver's *future sequence values* until the number of *sequence values* in the receiver's *past sequence values* is equal to `amount`.

If the number of *sequence values* in the receiver's *past sequence values* is equal to `amount` no action is taken.

Parameters

`amount` <integer> unspecified

Return Value

UNSPECIFIED

Errors

If `amount` is negative.

If the receiver has any *sequence values* and `amount` is greater than or equal to the total number of *sequence values* of the receiver.

5.9.1.6 Message: reset

Synopsis

Resets the position of the receiver to be at the beginning of the stream of values.

Definition: <sequencedStream>

Sets the receiver's *future sequence values* to be the current *past sequence values* appended with the current *future sequence values*. Make the receiver's *past sequence values* be empty.

Return Value

UNSPECIFIED

Errors

none

5.9.1.7 Message: setToEnd

Synopsis

Set the position of the stream to its end.

Definition: <sequencedStream>

All of the *receiver's future sequence values* are appended, in sequence, to the receiver's *past sequence values*. The receiver then has no *future sequence values*.

Return Value

UNSPECIFIED

Errors

none

5.9.2 Protocol: <gettableStream>

Conforms To

<Object>

Description

An object conforming to <gettableStream> can read objects from its *future sequence values*.

Messages

atEnd
do:
next
next:
nextLine
nextMatchFor:
peek
peekFor:
skip:
skipTo:
upTo:

5.9.2.1 Message: atEnd

Synopsis

Returns a Boolean indicating whether the receiver is at the end of its values.

Definition: <gettableStream>

Return *true* if the receiver has no *future sequence values* available for reading. Return *false* otherwise.

Return Value

<boolean> unspecified

Errors

none

5.9.2.2 Message: do: operation

Synopsis

Evaluates the argument with each receiver *future sequence value*, terminating evaluation when there are no more *future sequence values*.

Definition: <gettableStream>

Each member of the receiver's *future sequence values* is, in turn, removed from the future sequence values; appended to the past sequence values; and, passed as the argument to an evaluation of `operand`. The argument, `operation`, is evaluated as if sent the message `#value:`. The number of evaluations is equal to the initial size of the receiver's *future sequence values*. If there initially are no *future sequence values*, `operation` is not evaluated. The future sequence values are used as arguments in their sequence order. The result is undefined if any evaluation of `operand` changes the receiver's *future sequence values*

Parameters

`operation` <monadicValuable> uncaptured

Return Value

UNSPECIFIED

Errors

none

5.9.2.3 Message: next

Synopsis

Return the next object in the receiver.

Definition: <gettableStream>

The first object is removed from the receiver's *future sequence values* and appended to the end of the receiver's *past sequence values*. That object is returned as the value of the message. The returned object must conform to the receiver's *sequence value type*.

The result is undefined if there the receiver has no *future sequence values*.

Return Value

<Object> state

Errors

none

5.9.2.4 Message: next: amount

Synopsis

Returns a collection of the next `amount` objects in the stream.

Definition: <gettableStream>

A number of objects equal to `amount` are removed from the receiver's *future sequence values* and appended, in order, to the end of the receiver's *past sequence values.* A collection whose elements consist of those objects, in the same order, is returned. If amount is equal to 0 an empty collection is returned.

The result is undefined if `amount` is larger than the number of objects in the receiver's *future sequence values*.

Parameters

`amount` <integer> uncaptured

Return Value

<sequencedReadableCollection> new

Errors

`amount` < 0

5.9.2.5 Message: nextLine

Synopsis

Reads the next line from the stream.

Definition: <gettableStream>

Each object in the receiver's *future sequence values* up to and including the first occurrence of the objects that constitute an implementation defined end-of-line sequence is removed from the *future sequence values* and appended to the receiver's *past sequence values*. All of the transfered objects, except the end-of-line sequence objects, are collected, in order, as the elements of a string that is the return value. The result is undefined if there are no *future sequence values* in the receiver or if the future-sequence values do not include the end-of-line sequence.

Return Value

<readableString> new

Errors

If any of the *future sequence values* to be returned do not conform to the protocol <Character>.

5.9.2.6 Message: nextMatchFor: anObject

Synopsis

Reads the next object from the stream and returns *true* if the object is *equivalent* to the argument and *false* if not.

Definition: <gettableStream>

The first object is removed from the receiver's *future sequence value* and appended to the end of the receiver's *past sequence values*. The value that would result from sending #= to the object with `anObject` as the argument is returned.

The results are undefined if there are no *future sequence values* in the receiver.

Parameters

`anObject` <Object> uncaptured

Return Value

<boolean> unspecified

Errors

none

5.9.2.7 Message: peek

Synopsis

Returns the next object in the receiver's *future sequence values* without advancing the receiver's position. Returns *nil* if the receiver is at end of stream.

Definition: <gettableStream>

Returns the first object in the receiver's *future sequence values*. The object is not removed from the *future sequence values*. The returned object must conform to the receiver's *sequence value type*.

Returns *nil* if the receiver has no *future sequence values*. The return value will also be *nil* if the first future sequence object is *nil*.

Return Value

<Object> state

Errors

none

5.9.2.8 Message: peekFor: anObject

Synopsis

Peeks at the next object in the stream and returns *true* if it matches the argument, and *false* if not.

Definition: <gettableStream>

Returns the result of sending #= to the first object in the receiver's *future sequence values* with `anObject` as the argument. Returns *false* if the receiver has no *future sequence values*.

Parameters

`anObject` <Object> uncaptured

Return Value

<boolean> unspecified

Errors

none

5.9.2.9 Message: skip: amount

Synopsis

Skips the next `amount` objects in the receiver's *future sequence values*.

Definition: <gettableStream>

A number of objects equal to the lesser of `amount` and the size of the receiver's *future sequence values* are removed from the receiver's *future sequence values* and appended, in order, to the end of the receiver's *past sequence values***.**

Parameters

`amount` <integer> uncaptured

Return Value

UNSPECIFIED

Errors

none

5.9.2.10 Message: skipTo: anObject

Synopsis

Sets the stream to read the object just after the next occurrence of the argument and returns *true*. If the argument is not found before the end of the stream is encountered, *false* is returned.

Definition: <gettableStream>

Each object in the receiver's *future sequence values* up to and including the first occurrence of an object that is *equivalent* to `anObject` is removed from the *future sequence values* and appended to the receiver's *past sequence values*. If an object that is *equivalent* to `anObject` is not found in the receiver's *future sequence values*, all of the objects in *future sequence values* are removed from *future sequence values* and appended to *past sequence values*. If an object equivalent to anObject is not found *false* is returned. Otherwise return *true*.

Parameters

`anObject` <Object> uncaptured

Return Value

<boolean> unspecified

Errors

none

5.9.2.11 Message: upTo: anObject

Synopsis:

Returns a collection of all of the objects in the receiver up to, but not including, the next occurrence of the argument. Sets the stream to read the object just after the next occurrence of the argument. If the argument is not found and the end of the stream is encountered, an ordered collection of the objects read is returned.

Definition: <gettableStream>

Each object in the receiver's *future sequence values* up to and including the first occurrence of an object that is *equivalent* to `anObject` is removed from the *future sequence values* and appended to the receiver's *past sequence values*. A collection, containing, in order, all of the transferred objects except the object (if any) that is equivalent to `anObject` is returned. If the receiver's *future sequence values* is initially empty, an empty collection is returned.

Parameters

`anObject` <Object> uncaptured

Return Value

<sequencedReadableCollection> new

Errors

none

5.9.3 Protocol: <collectionStream>

Conforms To

<sequencedStream>

Description

An object conforming to <collectionStream> has a <sequencedReadableCollection> as its *stream backing store*.

Messages

contents

5.9.3.1 Message: contents

Synopsis

Returns a collection containing the complete contents of the stream.

Definition: <sequencedStream>

Returns a collection that contains the receiver's *past* and *future sequence values*, in order. The size of the collection is the sum of the sizes of the *past* and *future sequence values*.

Refinement: <collectionStream>

It is unspecified whether or not the returned collection is the same object as the backing store collection. However, if the returned collection is not the same object as *the stream backing store*

collection then the class of the returned collection is the same class as would be returned if the message `#select:` was sent to the backing store collection.

Return Value

<sequencedReadableCollection> unspecified

Errors

none

5.9.4 Protocol: <puttableStream>

Conforms To

<Object>

Description

An object conforming to <puttableStream> allows objects to be added to its *past sequence values*.

Messages

cr
flush
nextPut:
nextPutAll:
space
tab

5.9.4.1 Message: cr

Synopsis

Writes an end-of-line sequence to the receiver.

Definition: <puttableStream>

A sequence of character objects that constitute the implementation-defined end-of-line sequence is added to the receiver in the same manner as if the message `#nextPutAll:` was sent to the receiver with an argument string whose elements are the sequence of characters.

Return Value

UNSPECIFIED

Errors

It is erroneous if any element of the end-of-line sequence is an object that does not conform to the receiver's *sequence value type*.

5.9.4.2 Message: flush

Synopsis:

Update a stream's backing store.

Definition: <puttableStream>

Upon return, if the receiver is a *write-back stream,* the state of the *stream backing store* must be consistent with the current state of the receiver.

If the receiver is not a *write-back stream*, the effect of this message is unspecified.

Return Value

UNSPECIFIED

Errors

none

5.9.4.3 Message: nextPut: anObject

Synopsis

Writes the argument to the stream.

Definition: <puttableStream>

Appends `anObject` to the receiver's *past sequence values*. If the receiver's *future sequence values* is not empty, removes its first object.

Parameters

`anObject`	<Object>	captured

Return Value

UNSPECIFIED

Errors

It is erroneous if `anObject` is an object that does not conform to the receiver's *sequence value type*.

5.9.4.4 Message: nextPutAll: aCollection

Synopsis

Enumerates the argument, adding each element to the receiver

Definition: <puttableStream>

Has the effect of enumerating the `aCollection` with the message `#do:` and adding each element to the receiver with `#nextPut:`. That is,

```
aCollection do: [:each | receiver nextPut: each]
```

Parameters

`aCollection`	<collection>	uncaptured

Return Value

UNSPECIFIED

Errors

It is erroneous if any element of `aCollection` is an object that does not conform to the receiver's *sequence value type*.

5.9.4.5 Message: space

Synopsis

Writes a space character to the receiver.

Definition: <puttableStream>

The effect is the same as sending the message `#nextPut:` to the receiver with an argument that is the object that is the value returned when the message `#space` is sent to the standard global `Character`.

Return Value

UNSPECIFIED

Errors

It is erroneous if the space character is an object that does not conform to the receiver's *sequence value type*.

5.9.4.6 Message: tab

Synopsis

Writes a tab character to the receiver.

Definition: <puttableStream>

The effect is the same as sending the message `#nextPut:` to the receiver with an argument that is the object that is the value returned when the message `#tab` is sent to the standard global `Character`.

Return Value

UNSPECIFIED

Errors

It is erroneous if the tab character is an object that does not conform to the receiver's *sequence value type*.

5.9.5 Protocol: <ReadStream>

Conforms To

<gettableStream> <collectionStream>

Description

An object conforming to <ReadStream> has a positionable sequence of values that can be read. The *sequence values* are provided by a sequenced collection that serves as *the stream backing store*.

Messages

next:
upTo:

5.9.5.1 Message: next: amount

Synopsis

Returns a collection of the next `amount` objects in the stream.

Definition: <gettableStream>

A number of objects equal to `amount` are removed from the receiver's *future sequence values* and appended, in order, to the end of the receiver's *past sequence values.* A collection whose elements consist of those objects, in the same order, is returned. If amount is equal to 0 an empty collection is returned.

The result is undefined if `amount` is larger than the number of objects in the receiver's *future sequence values*.

Refinement: <ReadStream>

The result collection will conform to the same protocols as the object that would result if the message `#select:` was sent to the object that serves as the *stream backing store*.

Parameters

`amount` <integer> uncaptured

Return Value

<sequencedReadableCollection> new

Errors

amount < 0

5.9.5.2 Message: upTo: anObject

Synopsis:

Returns a collection of all of the objects in the receiver up to, but not including, the next occurrence of the argument. Sets the stream to read the object just after the next occurrence of the argument. If the argument is not found and the end of the stream is encountered, an ordered collection of the objects read is returned.

Definition: <gettableStream>

Each object in the receiver's *future sequence values* up to and including the first occurrence of an object that is *equivalent* to `anObject` is removed from the *future sequence values* and appended to the receiver's *past sequence values*. A collection, containing, in order, all of the transferred objects except the object (if any) that is equivalent to `anObject` is returned. If the receiver's *future sequence values* is initially empty, an empty collection is returned.

Refinement: <ReadStream>

The result collection will conform to the same protocols as the object that would result if the message `#select:` was sent to the object that serves as the *stream backing store*.

Parameters

`anObject` <Object> uncaptured

Return Value

<sequencedReadableCollection> new

Errors

none

5.9.6 Protocol: <WriteStream>

Conforms To

<puttableStream> <collectionStream>

Description: <WriteStream>

An object conforming to <WriteStream> has a positionable sequence of values to which new values may be written. The initial *sequence values* are provided by a collection that serves as *the stream backing store*. It is implementation defined whether a <WriteStream> is a *write-back stream*. Even if a <WriteStream> is not a *write-back stream*, its associated collection may be subject to modification in an unspecified manner as long as it is associated with the stream.

Messages

none

5.9.7 Protocol: <ReadWriteStream>

Conforms To

<ReadStream> <WriteStream>

Description

An object conforming to <ReadWriteStream> can read from its *future sequence values* or write to its *past sequence values*. The *sequence values* are provided by a collection that serves as *the stream backing store*. It is implementation defined whether a <ReadWriteStream> is *a write-back stream*. Even if a <ReadWriteStream> is not a *write-back stream*, its associated collection may be subject to modification in an unspecified manner as long as it is associated with the stream.

Messages

none

5.9.8 Protocol: <Transcript>

Conforms To

<puttableStream>

Description

An object conforming to <Transcript> is a <puttableStream> for logging status messages from Smalltalk programs. The *sequence value type* of <Transcript> is <Character>. There may be an implementation defined *stream backing store* that receives characters written to the stream in an implementatiuon defined manner.

Standard Globals

Transcript	Conforms to the protocol <Transcript>. Its language element type is unspecified. This is a <Transcript> that is always available to output textual messages in an implementtion defined manner.

Messages

none

5.9.9 Protocol: <ReadStream factory>

Conforms To

<Object>

Description

<ReadStream factory> provides for the creation of objects conforming to the <ReadStream> protocol whose sequence values are supplied by a collection.

Standard Globals

`ReadStream`	Conforms to the protocol <ReadStream factory>. Its language element type is unspecified. This is a factory for streams that conform to <ReadStream>.

Messages

on:

5.9.9.1 Message: on: aCollection

Synopsis

Returns a stream that reads from the given collection.

Definition: <ReadStream factory>

Returns an object conforming to <ReadStream> whose *future sequence values* initially consist of the elements of `aCollection` and which initially has no *past sequence values*. The ordering of the *sequence values* is the same as the ordering used by #do: when sent to `aCollection`. The *stream backing store* of the returned object is `aCollection`.

Parameters

`aCollection`	<sequencedReadableCollection>	captured

Return Value

<ReadStream>	new

Errors

none

5.9.10 Protocol: <ReadWriteStream factory>

Conforms To

<Object>

Description

<ReadWriteStreamfactory> provides for the creation of objects conforming to the <WriteStream> protocol whose sequence values are supplied by a collection.

Standard Globals

`ReadWriteStream`	Conforms to the protocol <ReadWriteStream factory>. Its language element type is unspecified. This is a factory for streams that conform to <ReadWriteStream>.

Messages

with:

5.9.10.1 Message: with: aCollection

Synopsis

Returns a stream that reads the elements of the given collection and can write new elements.

Definition: <ReadWriteStream factory>

Returns an object conforming to <ReadWriteStream> whose *past sequence values* initially consist of the elements of `aCollection` and which initially has no *future sequence values*. The ordering of the *sequence values* is the same as the ordering used by #do: when sent to `aCollection`. The *stream backing store* of the returned object is `aCollection`. The *sequence value type* of the write stream is the element type of `aCollection`. Any restrictions on objects that may be elements of `aCollection` also apply to the stream's *sequence elements*.

Parameters

`aCollection` <sequencedCollection> captured

Return Value

<ReadWriteStream> new

Errors

none

5.9.11 Protocol: <WriteStream factory>

Conforms To

<Object>

Description

<WriteStream factory> provides for the creation of objects conforming to the <WriteStream> protocol whose sequence values are supplied by a collection.

Standard Globals

`WriteStream`	Conforms to the protocol <WriteStream factory >. Its language element type is unspecified. This is a factory for streams that conform to <WriteStream>.

Messages

with:

5.9.11.1 Message: with: aCollection

Synopsis

Returns a stream that appends to the given collection.

Definition: <WriteStream factory>

Returns an object conforming to <WriteStream> whose *past sequence values* initially consist of the elements of `aCollection` and which initially has no *future sequence values*. The ordering of the *sequence values* is the same as the ordering used by `#do:` when sent to `aCollection`. The *stream backing store* of the returned object is `aCollection`. The *sequence value type* of the write stream is the element type of `aCollection`. Any restrictions on objects that may be elements of `aCollection` also apply to the stream's *sequence elements*.

Parameters

`aCollection` <sequencedCollection> captured

Return Value

<WriteStream> new

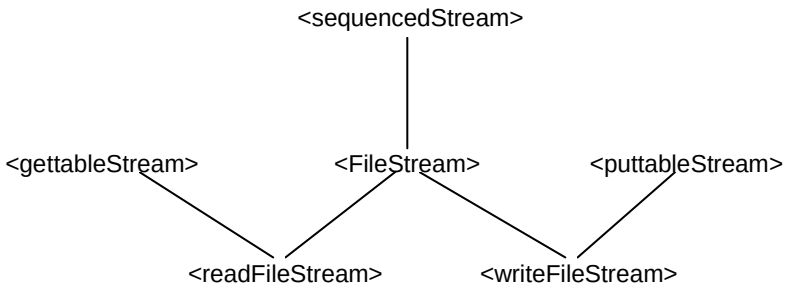
Errors

none

5.10 File Stream Protocols

This section includes protocols that define the behavior of streams that access the contents of files.

The graph below shows the conformance relationships between the protocols defined in this section (except for <puttableStream>, <gettableStream>, and <sequencedStream>, which are contained in the section on stream protocols).



<FileStream factory>

5.10.1 Protocol: <FileStream>

Conforms To

<sequencedStream>

Description

Provides protocol for streams over external files. The external file serves as *the stream backing store*. When objects are read or written from a file stream they must be translated from or two an external data representation. File streams have an *external stream type* that is specified when the stream is created. The *external stream type* defines the data translation and the *sequence value type* for the stream. *External stream types* are specified using <symbol> objects. The standard defines the following *external stream types*:

`#'binary'`	The external data is treated as sequence of 8-bit bytes. The *sequence value type* is <integer> with values restricted to the range 0 to 255.
`#'text'`	The external data is treated as a sequenced of 8-bit characters encoded using an implementation defined external character set. The *sequence value type* is <Character> restricted to those specific characters that may be represented in the external character set.

Implementations may define other *external stream types*.

Rational

The file stream capability specified in the standard was motivated by the desire to support a useful, yet minimal set of functionality and to take as a guide (i.e. subset) the Posix standard.

There is specification only for the creation and use of readable and writeable file streams. There is not support for read/write file streams. Nor is there any specification of file or directory manipulation, as these facilities are considered by the Committee to be too platform-dependent and too implementation-dependent to standardize at this time, and it is felt that streaming is adequate.

In addition, we only support the most common subset of the Posix file stream creation modes, rather than the full set.

We also considered the tradeoffs of specifying a wide range of creation messages, but decided that one fully-functional message and one most-typical creation message for each of read and write file streams would be adequate. Implementations are not prohibited from providing more options.

Messages

contents
externalType
isBinary
isText

5.10.1.1 Message: contents

Synopsis

Returns a collection containing the complete contents of the stream.

Definition: <sequencedStream>

Returns a collection that contains the receiver's *past* and *future sequence values*, in order. The size of the collection is the sum of the sizes of the *past* and *future sequence values*.

Refinement: <FileStream>

If the *external stream type* is #'binary' the returned collection conforms to <ByteArray>. If the *external stream type* is #'text' the returned collection conforms to <String>.

Return Value

<ByteArray> new

<String> new

Errors

None

5.10.1.2 Message: externalType

Synopsis

Returns a symbol that identifies the *external stream type* of the receiver.

Definition: <FileStream>

Return the symbol that identifies the *external stream type* of the receiver.

Return Value

<symbol> unspecified

Errors

none

5.10.1.3 Message: isBinary

Synopsis:

Answer wthether the receiver's data is binary.

Definition: <FileStream>

Answer *true* if the *sequence value type* conforms to <integer>. Otherwise answer *false*.

Return Value

<boolean> unspecified

Errors

none

5.10.1.4 Message: isText

Synopsis:

Answer whether the receiver's data is characters.

Definition: <FileStream>

Answer *true* if the *sequence value type* conforms to <Character>. Otherwise answer *false*.

Return Value

<boolean> unspecified

Errors

none

5.10.2 Protocol: <readFileStream>

Conforms To

<FileStream> <gettableStream>

Description

Provides protocol for traversing and reading elements in an external file. The *sequence values* are provided by the external file which also serves as *the stream backing store*.

Messages

next:
upTo:

5.10.2.1 Message Refinement: next: amount

Synopsis

Returns a collection of the next `amount` objects in the stream.

Definition: <gettableStream>

A number of objects equal to `amount` are removed from the receiver's *future sequence values* and appended, in order, to the end of the receiver's *past sequence values.* A collection whose elements consist of those objects, in the same order, is returned. If amount is equal to 0 an empty collection is returned.

The result is undefined if `amount` is larger than the number of objects in the receiver's *future sequence values*.

Refinement: <readFileStream>

The result collection will conform to the same protocols as the object that would result if the message `#contents` was sent to the receiver.

Parameters

`amount` <integer> uncaptured

Return Value

<sequencedReadableCollection> new

Errors

amount < 0

5.10.2.2 Message Refinement: upTo: anObject

Synopsis:

Returns a collection of all of the objects in the receiver up to, but not including, the next occurrence of the argument. Sets the stream to read the object just after the next occurrence of the argument. If the argument is not found and the end of the stream is encountered, an ordered collection of the objects read is returned.

Definition: <gettableStream>

Each object in the receiver's *future sequence values* up to and including the first occurrence of an object that is *equivalent* to `anObject` is removed from the *future sequence values* and appended to the receiver's *past sequence values*. A collection, containing, in order, all of the transferred objects except the object (if any) that is equivalent to `anObject` is returned. If the receiver's *future sequence values* is initially empty, an empty collection is returned.

Refinement: <readFileStream>

The result collection will conform to the same protocols as the object that would result if the message `#contents` was sent to the receiver.

Parameters

`anObject` <Object> uncaptured

Return Value

<sequencedReadableCollection> new

Errors

none

5.10.3 Protocol: <writeFileStream>

Conforms To

<FileStream> <puttableStream>

Description

Provides protocol for storing elements in an external file. The *sequence values* are provided by the external file which also serves as *the stream backing store*. A <writeFileStream> is a *write-back stream*.

Messages

none

5.10.4 Protocol: <FileStream factory>

Conforms To

<Object>

Description

<FileStream factory> provides for the creation of objects conforming to the <readFileStream> or <writeFileStream> protocols.

Standard Globals

`FileStream` Conforms to the protocol <FileStream factory>. Its program element type is unspecified. This is a factory for collections that conform to <readFileStream> and <writeFileStream>.

Messages

read:
read:type:
write:
write:mode:
write:mode:check:type:

5.10.4.1 Message: read: fileId

Synopsis

Returns a read file stream that reads text from the file with the given name.

Definition: <FileStream factory>

The result is the same as if the message `#read:type:` was sent to the receiver with `fileId` as the first argument and the symbol `#'text'` as the second argument.

Parameters

`aString`<String> unspecified

Return Value

<readFileStream> new

Errors

As defined by <FileStream factory> #read:type:

5.10.4.2 Message: read: fileId type: fileType

Synopsis

Returns a read file stream that reads from the file with the given name.

Definition: <FileStream factory>

Locate an external file that is identified by the value of `fileID`. The syntax of the `fileID` string is implementation defined.

Return an object conforming to <readFileStream> whose *future sequence values* initially consist of the elements of the external file and which initially has no *past sequence values*. The ordering of the *sequence values* is the same as the ordering within the external file. The external file serves as the *stream backing store* of the returned object. The value of `fileType` determines the *external stream type* and *sequence value type* of the result object.

Parameters

`fileId` <readableString> unspecified

`fileType`<symbol> unspecified

Return Value

<readFileStream> new

Errors

It is an error if the file does not exist, or if the user does not have read access to the file.

5.10.4.3 Message: write: fileId

Synopsis

Returns a write file stream that writes text to the file with the given name.

Definition: <FileStream factory>

The result is the same as if the message `#write:mode:check:type:` was sent to the receiver with `fileId` as the first argument, `#'create'` as the second argument, *false* as the third argument, and the symbol `#'text'` as the fourth argument.

Parameters

`fileId` <readableString> unspecified

Return Value

<writeFileStream> new

Errors

As defined by <FileStream factory> #write:mode:check:type:

5.10.4.4 Message: write: fileId mode: mode

Synopsis

Returns a write file stream that writes text to the file with the given name.

Definition: <FileStream factory>

The result is the same as if the message `#write:mode:check:type:` was sent to the receiver with `fileId` as the first argument, `mode` as the second argument, *false* as the third argument, and the symbol `#'text'` as the fourth argument.

Parameters

`fileId`	<readableString>	unspecified
`mode`	<symbol>	unspecified

Return Value

<writeFileStream> new

Errors

As defined by <FileStream factory> #write:mode:check:type:

5.10.4.5 Message: write: fileId mode: mode check: check type: fileType

Synopsis

Returns a write file stream that writes to the file with the given name.

Definition: <FileStream factory>

Depending upon the values of `check` and `mode`, either create a new external file or locate an existing external file that is identified by the value of `fileID`. The syntax of the `fileID` string is implementation defined.

Return an object conforming to <writeFileStream>. The external file serves as the *stream backing store* of the returned object. The returned object is a *write-back stream*. The value of `fileType` determines the *external stream type* and *sequence value type* of the result object.

Valid values for mode are: `#'create'`, `#'append'`, and `#'truncate'`. The meaning of these values are:

`#'create'`	create a new file, with initial position at the beginning
`#'append'`	use an existing file, with initial position at its end
`#'truncate'`	use an existing file, initially truncating it.

The value of `mode` determines the initial state of the *past sequence values* and *future sequence values* of the result object. If `mode` is `#'create'` or `#'truncate'` the *past sequence values* and *future sequence values* are both initially empty. If mode is `#'append'` the *past sequence values* initially consist of the elements of the external file and *future sequence values* is initially empty. The ordering of the *sequence values* is the same as the ordering within the external file.

The `check` flag determines whether the file specified by `fileID` must exist or not exist.

If `mode` = `#'create'` and `check` = *false* and the file exists, then the existing file is used.

If `mode` = `#'append'` and `check` = *false* and the file does not exist, then it is created.

If `mode` = `#'truncate'` and `check` = *false* and the file does not exist, then it is created.

This operation is undefined if a value other than `#'create'`, `#'append'` or `#'truncate'` is used as the `mode` argument.

Parameters

`fileID`	<readableString>	unspecified
`mode`	<symbol>	unspecified
`check`	<boolean>	unspecified
`fileType`	<symbol>	unspecified

Return Value

<writeFileStream> new

Errors

If `mode` = #create and `check` = *true* and the file exists.

If `mode` = #append and `check` = *true* and the file does not exist.

If `mode` = #truncate and `check` = *true* and the file does not exist.

If the user does not have write permissions for the file.

If the user does not have creation permissions for a file that is to be created.

6. Glossary

The Smalltalk standard defines and uses the following terms:

abnormal termination Termination of a block evaluation in any manner that would not have resulted in the normal return of a result from a `#value` message if that message had been used to initiate the evaluation. Abnormal termination occurs when code in a block executes a return statement or when any action external to the block permanently and irrevocably terminates evaluation of the block.

advance To move a stream forward. Reading from a <readableStream> adds the read object to the stream's past sequence values and removes it from the stream's future sequence values.

argument An object encapsulated in a message that is required by the receiver to perform the operation being requested.

array A data structure whose elements are associated with integer indices.

assignment An expression describing a change of a variable's value.

binary message A message with one argument whose selector is made up of one or two special characters.

bind To cause a variable to refer to an object.

block Certain valuables called blocks have their evaluation rules determined by the syntax and semantics of the Smalltalk language. For details of how blocks are evaluated in context, refer to the Block Syntax section of the specification.

block argument A parameter that must be supplied when certain *blocks* are evaluated.

cascading A description of several messages to one object in a single expression.

class definition The Smalltalk language construct that defines the representation and behavior of instance objects and a globally named *class object* that implements the class behavior.

class object An object defined by a *class definition* that responds to the class messages and which has a global name binding.

closure The result of evaluating a block; the representation of the context of execution of all enclosing blocks.

comparable Two objects are comparable if there is an ordering defined between them. While there is no precise way to define which objects have such

an ordering, it is generally the case that such objects must both conform to another common protocol in addition to <magnitude>. For example, all objects that conform to the protocol <number> are comparable.

context The values of variables defined within a block during a particular execution of the method represented by that block.

default action The method that is executed in response to an exception if the current *exception environment* does not contain an *exception handler* that *handles* the exception.

element An object is an element of a collection if the object will be passed as an argument to the argument of the message `#do:`.

element type A set of acceptable objects for *elements* of a collection. Unless otherwise specified, the *element type* of a collection is <Object>.

equivalent Two objects are considered equivalent if the result of sending the message #= to one of the objects with the other object as the argument returns *true.*

evaluation context The stack of suspended method and block activations that represents the continuation at a point of execution in the program.

exception action The object conforming to the protocol <valuable> that will be evaluated if its containing *exception handler* is selected to service an exception.

exception environment An abstract entity that is a LIFO list of *exception handlers.* An *exception environment* may be logically searched starting from the most recently "pushed" *exception handler.*

exception handler An abstract entity that associates an *exception selector* with an *exception action* for the duration of a *protected block.* During the evaluation of the *protected block,* occurrence of an exceptional condition that matches the *exception selector* will result in the execution of the *exception action.* An exception handler is established by sending the message `#on:do:` to the *protected block* with the *exception selector* as the first argument and the *exception action* as the second argument.

exception selector An object conforming to the protocol <exceptionSelector> that is contained in an *exception handler* and used to determine whether the handler should be used to service an exception.

expression A sequence of characters that describes an object.

false The value of the reserved identifier "false".

fragile The implementation of a class's behavior is fragile if it is possible for method in a subclass by the mere fact of its existence to inadvertently

cause methods inherited from the class to malfunction. Implementation may use underscore prefixed method selectors or other implementation specific means to implement classes in a non-fragile manner.

future sequence values The *sequence values* yet to be read by a stream.

general subclass Any class that either directly or indirectly inherits from a superclass is a general subclass of the superclass.

handle An *exception handler* is said to handle an exception if its *exception selector* will respond with *true* if asked if it should service the exception.

handler block A block that is specified as an *exception action.*

handler environment The state of the current *exception environment* as it existed immediately before the execution of the `#on:do:` message that establishes a new *exception handler.*

hash value The non-negative integer result of sending the message `#hash` to an object.

identical Two objects are considered identical if they are the same object. In other workds, the result of sending the message #== to one of the objects with the other object as the argument is *true.*

identifier A lexical representation for variables and selectors.

identity hash value The non-negative integer result of sending the message `#identityHash` to an object.

identity object An object defined such that a=b implies a==b

immutable object An object whose state cannot be modified.

key A key is an object used to selectively access a single *element* of a collection. Not all collections support the use of keys to access of their *elements.*

key equivalence The operation used to compare keys in a dictionary. Protocols that refine <abstractDictionary> must define the meaning of this term.

key lookup Lookup of a key in a dictionary using *key equivalence.*

keyword An identifier with a trailing colon.

keyword message A message with one or more arguments whose selector is made up of one or more keywords.

lexical order	Ordering two sequences of values by comparing their elements in order. The first two elements that differ determine the order.
literal	An expression describing a constant, such as a number or string.
local time	A system of measuring and describing time. Local times specify the abbreviations, names, and numberings for various components of a date time.
message argument	An object that specifies additional information for an operation.
message selector	The name of the type of operation a message requests of its receiver.
method	The executable representation of an operation. It consists of zero or more parameters and a number of expressions that are evaluated sequentially.
nil	The value of the reserved identifier "nil".
numeric representation	The numeric representation of a numeric object is an implementation dependent representation of a set of numbers conforming to a specific protocol. A numeric representation may include limits on precision and range of its values.
past sequence values	The *sequence values* already read, written, or skipped by a stream.
precision	The precision of a numeric representation is the number of significant digits in the representation.
program	A description of the data and operations that comprise a computation.
protected block	An object conforming to the protocol <valuable> that is the scope over which an *exception handler* is active.
pseudo variable name	An *expression* similar to a variable name. However, unlike a variable name, the value of a *pseudo variable name* cannot be changed by an assignment.
range	The range of a numeric representation is the set of number between the upper and lower bounds.
receiver	The object to which a message is sent.
resumable	An exception for which it is possible to resume execution from the point at which the exception was signaled.

resumption value The value that is returned to the signaler from the *exception action* of a *resumable* exception.

scope The mechanism by which the language restricts the visibility of variables. A name can be declared to have local scope within a block or method. Scopes can be nested. A name declared local to a scope represents the same entity within that scope, and all scopes nested within it.

sequence value A value in a stream.

signaled exception During the signaling of an exception, the object conforming to the protocol <exceptionDescription> that describes the exception and which is used to select an *exception handler.*

signaling environment The state of the current *exception environment* at the time that an exception is signaled.

sort block A <dyadicValuable> object used by <SortedCollection> objects to order their elements. The *sort block* must return a <boolean> result.

stream backing store An object or external data store which provides or receives the sequence values of the associated stream.

subexception An exception that is a specialization of another exception. An *exception handler* for the more general exception will also serve as an *exception handler* for the subexception.

symbol A string whose sequence of characters is guaranteed to be different from that of any other symbol.

temporally invariant A message is temporally invariant if the repeated application of the message to identical receivers and arguments will always yield an equivalent result.

true The value of the reserved identifier "true".

unary message A message without arguments.

unbounded precision A numeric representation has unbounded precision if it can precisely represent all numbers conforming to its protocol.

value The result of evaluating an object conforming to the protocol <valuable>.

variable name An expression describing the current value of a variable.

white space Characters that serve as token separators; ignored in a program parse.

write-back stream A stream that supports the writing of objects and that has a *stream backing store* that receives the objects written to the stream. A buffer may exist between a *write-back stream* and its *stream backing store* and there may be a latency between the time an object is written to the stream and its appearance in the backing store.

7. Index of Protocols

T

V

W

Z

8. References

[Goldberg83] Goldberg, A. & Robson, D. (1983). Smalltalk-80: The language and its implementation. Addison-Wesley.

[Kay93] Kay, A.C. (1993). The Early History of Smalltalk. ACM SIGPLAN Notices, v. 28, n. 3, March.

[Thomson93] Thomson, David G. [1993]. Believable Specifications: Organizing and Describing Object Interfaces Using Protocol Conformance. Master Thesis, Carleton University, Ottawa, Ontario.