The Slate Programmer's Reference Manual

Brian T. Rice, Lee Salzman, Timmy Douglas

November 13, 2020

Contents

1 Introduction **5**
1.1 Conventions . . . 5
1.2 Terms . . . 5

2 Language Reference **6**
2.1 Objects . . . 6
2.1.1 Code Blocks . . . 6
2.1.2 Slots . . . 8
2.1.3 Inheritance . . . 8
2.2 Sending Messages . . . 9
2.2.1 Unary Messages . . . 9
2.2.2 Binary Messages . . . 10
2.2.3 Keyword Messages . . . 11
2.2.4 Implicit-context Messages . . . 11
2.3 Sequencing Expressions . . . 12
2.4 Methods . . . 13
2.4.1 Roles . . . 13
2.4.2 Method Definitions . . . 13
2.4.3 Expression-based Definitions . . . 14
2.4.4 Lookup Semantics . . . 15
2.4.5 Optional Keyword Arguments . . . 16
2.4.6 Resends . . . 17
2.4.7 Subjective Dispatch . . . 18
2.5 Type Annotations . . . 19
2.6 Macro Message-sends . . . 19
2.6.1 Defining new Macro-methods . . . 20

2.6.2 Quoting and Unquoting 20
2.6.3 Message Cascading 21
2.6.4 Slots as Block Variables 22
2.6.5 Expression Substitution 23
2.6.6 Source Pattern Matching 23
2.7 Literals 23
2.7.1 Numbers 23
2.7.2 Characters 23
2.7.3 Strings 24
2.7.4 Symbols 24
2.7.5 Arrays 25
2.7.6 Blocks 26

3 The Slate World **26**
3.1 Overall Organization 26
3.1.1 The lobby 26
3.1.2 Naming and Paths 27
3.2 Core Behaviors 27
3.2.1 Default Object Features 28
3.2.2 Oddballs 29
3.3 Introspection 29
3.3.1 Slot Queries 29
3.3.2 Method Queries 30
3.4 Traits 31
3.5 Control-flow 32
3.5.1 Boolean Logic 32
3.5.2 Basic Conditional Evaluation 32
3.5.3 Early Returns 33
3.5.4 Looping 34
3.5.5 Method Operations 35
3.6 Symbols 35
3.7 Magnitudes and Numbers 36
3.7.1 Basic Types 36
3.7.2 Basic Operations 36
3.7.3 Non-core Operations 37
3.7.4 Limit Numerics 38
3.7.5 Dimensioned Units 38
3.8 Collections 38

3.8.1 Extensible Collections . . . 40
3.8.2 Sequences . . . 40
3.8.3 Strings and Characters . . . 42
3.8.4 Collections without Duplicates . . . 43
3.8.5 Mappings and Dictionaries . . . 43
3.8.6 Linked Collections . . . 43
3.8.7 Vectors and Matrices . . . 44
3.9 Streams and Iterators . . . 44
3.9.1 Basic Protocol . . . 44
3.9.2 Basic Stream Variants . . . 45
3.9.3 Basic Instantiation . . . 47
3.9.4 Collecting Protocols . . . 48
3.9.5 Iterator Streams . . . 48
3.10 External Resources . . . 49
3.10.1 Consoles . . . 50
3.10.2 Files . . . 51
3.10.3 Shells and Pipes . . . 51
3.10.4 Networking . . . 52
3.11 Paths . . . 53
3.12 Exceptions . . . 53
3.12.1 Types . . . 54
3.12.2 Protocol . . . 54
3.13 Concurrency . . . 55
3.13.1 Processes . . . 55
3.13.2 Promises and Eventual-Sends . . . 55
3.14 Types . . . 55
3.14.1 Types . . . 56
3.14.2 Operations . . . 56
3.14.3 Type Annotations . . . 56
3.14.4 Type Inference . . . 57
3.15 Modules . . . 57
3.15.1 Types . . . 57
3.15.2 Operations . . . 58
3.15.3 Auto-loading . . . 58
3.16 Persistence . . . 59
3.16.1 Slate Heap Images . . . 59
3.16.2 Heap Image Segments . . . 60

4 Style Guide **60**
4.1 Environment organization 60
4.1.1 Namespaces 60
4.1.2 Exemplars or Value Objects 60
4.2 Naming Methods 60
4.2.1 Attributes 61
4.2.2 Queries 61
4.2.3 Creating 61
4.2.4 Performing Actions 61
4.2.5 Binary Operators 62
4.3 Instance-specific Dispatch 62
4.3.1 Motivations 62
4.3.2 Limitations 63
4.4 Organization of Source 63
4.5 Type-Annotating Expressions 63
4.6 Writing Test Cases 63

List of Figures

1 Core Object Inheritance 27
2 Core Collections Inheritance 39
3 Stream Inheritance 46

1 Introduction

Slate is a member of the Smalltalk family of languages which supports an object model in a similar prototype-based style as Self[Ungar et al 95], extended and reshaped to support multiple-dispatch methods. However, unlike Self, Slate does not rely on a literal syntax that combines objects and blocks, using syntax more akin to traditional Smalltalk. Unlike a previous attempt at providing prototype-based languages with multiple dispatch[Chambers 97], Slate is dynamic and more free-form in style, supporting the simplicity and flexibility of syntax and environment of the Smalltalk family. It is intended that both Smalltalk and Self styles of programs can be ported to Slate with minimal effort. Finally, Slate contains extensions including optional keywords, optional type-declarations, subjective dispatch and syntactic macros, that can be used to make existing programs and environment organizations much more powerful than in traditional object-based programming.

1.1 Conventions

Throughout this manual, various terms will be highlighted in different ways to indicate the type of their significance. If some concept is a certain programming utility in Slate with a definite implementation, it will be formatted in a `typewriter-style`. If a term is technical with a consistent definition in Slate, but cannot have a definite implementation, it will be set in SMALL CAPITAL LETTERS. Emphasis on its own is denoted by *italics*. When expression/result patterns are entered, typewriter-style text will be used with a `Slate>` prompt before the statement and its result will be set in *`italicized typewritten text`* below the line.

1.2 Terms

Slate is an object-oriented language, and as such works with some terms worth describing initially for clarity. These are primarily inspired by the metaphor of computational entities which communicate via messages, as follows:

Object is some thing in the system that can be identified.

Method is some behavior or procedure that is defined on some objects or class of objects.

Message is the act of requesting a behavior or procedure from some objects, the message's *arguments*. The requestor is known as the *sender*.

Answer is the response to a message; a value that expressions evaluate into or return to the message's sender.

Selector is the name of a method or a message-send.

Inheritance is a relationship between objects that confers one object's (the parent) behavior on another (the child).

Dispatch is the process of determining, from a message-send, what method is appropriate to invoke to implement the behavior. This is also referred to as *lookup*.

2 Language Reference

2.1 Objects

OBJECTS are fundamental in Slate; everything in a running Slate system consists of objects. Slate objects consist of a number of slots and roles: slots are mappings from `Symbol`s to other objects, and roles are a means of organizing code that can act on the object. Slots themselves are accessed and updated by a kind of message-send which is not distinguishable from other message-sends syntactically, but have some important differences.

Objects in Slate are created by *cloning* existing objects, rather than instantiating a class. When an object is cloned, the created object has the same slots and values as the original one. The new object will also have the access and update methods for those slots carried over to the new object. Other methods defined on the object will propagate through an analogue of a slot called a role, explained in section 2.4 on Methods.

Both control flow and methods are implemented by specialized objects called blocks, which are code closures. These code closures contain their own slots and create activation objects to handle run-time context when invoked. They can also be stored in slots and sent their own kinds of messages.

2.1.1 Code Blocks

A code block is an object representing an encapsulable context of execution, containing local variables, input variables, the capability to execute expressions sequentially, and finally answers a value to its point of invocation. The default return value for a block is the last expression's value; an early return via `^` (see 3.5.3 on page 33) can override this.

Blocks have a special syntax for building them up syntactically. Block expressions are delimited by square brackets. The block's optional header can specify input and local slots between vertical bars (`||`), and then a sequence of expressions which comprises the block's body. The input syntax allows specification of the slot names desired at the beginning. For example,

```
Slate> [| :arg num | num: 4. arg + num].
[]
```

creates and returns a new block. Within the header, identifiers that begin with a colon such as `:arg` above are parsed as input slots. The order in which they are specified is the order that arguments matching them must be passed in later to evaluate the block. If the block is evaluated later, it will return the expression after the final stop (the period) within the brackets, `arg + num`. In this block, `arg` is an input slot, and `num` is a local slot which is assigned to and then used in a following expression. The order of specifying the mix of input and local slots does not affect the semantics, but the order of the input slots directly determines what order arguments need to be passed to the block to assign them to the correct slots.

Using the term "slot" for local and input variables is not idle: the block is an actual object with slots for each of these variables, and accessors defined on them which are even callable from outside the block, considering it as an object.

In order to invoke a block, the caller must know how many and in what order it takes input arguments. Every block responds to `applyTo:`, which takes an array of the input values as its other argument. The block is immediately evaluated, and the result of the evaluation is the block's execution result. For example,

```
Slate> [| :x :y | x quo: y] applyWith: 17 with: 5.
3
Slate> [| :a :b | (a raisedTo: 2) - (4 * b)]
 applyTo: {3. 4}.
-7
```

Arguments can also be passed in using some easier messages. Blocks that don't expect any inputs respond to `do`, as follows:

```
Slate> [| a b | a: 4. b: 5. a + b] do.
9
```

Blocks that take one, two, or three inputs, each have special messages `applyWith:`, `applyWith:with:`, and `applyWith:with:with:` which pass in the inputs in the order they were declared in the block header.

If a block is empty, contains an empty body, or the final expression is terminated with a period, it returns `Nil` when evaluated:

```
Slate> [] do.
Nil
Slate> [| :a :b |] applyTo: {0. 2}.
Nil
Slate> [3. 4.] do.
Nil
```

If more arguments are passed than the block expects as inputs, an error is raised unless the block specifies a single "rest" argument parameter, using an asterisk prefix for the argument identifier in its header (e.g. `*rest`), in which case the slot is bound to an array of the inputs not otherwise bound.

Blocks furthermore have the property that, although they are a piece of code and the values they access may change between defining the closure and invoking it, the code will "remember" what objects it depends on, regardless of what context it may be passed to as a slot value. It is called a lexical closure since it "closes over" the environment and variables used in its definition, the lexical context where it was born. This is critical for implementing good control structures in Slate, as is explained later. Basically a block is an activation of its code composed with an environment that can be saved and invoked (perhaps multiple times) long after it is created, and always do so in the way that it reads where it was defined.

2.1.2 Slots

Slots may be mutable or immutable, and explicit slots or delegation (inheritance) slots. These four possibilities are covered by primitive methods defined on all objects.

Slate provides several primitive messages to manage DATA SLOTS (or *non-delegating* slots):

obj addSlot: slotSymbol adds a slot using the `Symbol` as its name, initialized to `Nil`.

obj addSlot: slotSymbol valued: val adds a slot under the given name and initializes its value to the given one.

addImmutableSlot:valued: performs the same as the above method, only without installing a mutator method.

obj removeSlot: slotSymbol removes the slot with the given name on the object directly and returns whatever value it had.

The effect of all of the slot addition methods when a slot of the same name is present is to update the value and attributes of the slot rather than duplicate or perform nothing.

2.1.3 Inheritance

Slate's means of sharing and conferring behavior involves the use of a DELEGATION ARRAY, whereby each object can name and access each object it inherits from.

The delegates' inheritance role involves a precedence order which affects the lookup of messages; from the object's perspective, the delegate slot array goes from least important to most important. Section 2.4.4 on page 15 explains the details of lookup semantics.

The relevant primitives specific to delegation slots are:

obj addDelegate: val and **obj insertDelegate: val** add a delegate. The delegates array for an object is stored in the object's map.

_delegates `and` **_delegates:** allow for finer control over manipulating delegates, since they explicitly set the delegate array. Use with care.

2.2 Sending Messages

Expressions in Slate mainly consist of messages sent to argument objects. The left-most argument is not considered an implicit receiver as it is with most message-passing languages, however. This means that when identifying a variable, it is really a message that is being sent to the context or argument.

An important issue is that every message selector (and slot name) is *case-sensitive* in Slate, that is, there is a definite distinction between what `AnObject`, `anobject`, and `ANOBJECT` denote even in the same context. Furthermore, the current implementation is *whitespace-sensitive* as well, in the sense that whitespace must be used to separate messages in order for them to be considered separate. For example, `ab+4` will be treated as one message, but `ab + 4` is a send of a binary message.

There are three basic types of messages, with different syntaxes and associativities: unary, binary, and keyword messages. *Precedence* is determine entirely by the syntactic form of the expression, but it can of course be overridden by enclosing expressions in parentheses. An implicit left-most argument can be used with all of them. The default precedence for forms is as follows:

1. Literal syntax: arrays, blocks, block headers, statement sequences.
2. Unary messages.
3. Binary messages.
4. Keyword messages.

A concept that will be often used about message-sends is that of the name of a message, its SELECTOR. This is the `Symbol` used to refer to the message or the name of a method that matches it. Slate uses three styles of selectors, each with a unique but simple syntax.

2.2.1 Unary Messages

A UNARY MESSAGE does not specify any additional arguments. It is written as a name following a single argument; it has a post-fix form.

Some examples of unary message-sends to explicit arguments include:

```
Slate> 42 print.
'42'
Slate> 'Slate' clone.
'Slate'
```

Unary sends associate from left to right. So the following prints the factorial of 5:

```
Slate> 5 factorial print.
'120'
```

Which works the same as:

```
Slate> (5 factorial) print.
'120'
```

Unary selectors can be most any alpha-numeric identifier, and are identical lexically to ordinary identifiers of slot names. This is no coincidence, since slots are accessed via a type of unary selector.

2.2.2 Binary Messages

A BINARY MESSAGE is named by a special non-alphanumeric symbol and 'sits between' its two arguments; it has an infix form. Binary messages are also evaluated from left to right; there is no special *precedence* difference between any two binary message-sends.[1]

These examples illustrate the precedence and syntax:

```
Slate> 3 + 4.
7
Slate> 3 + 4 * 5.
35
Slate> (3 + 4) * 5.
35
Slate> 3 + (4 * 5).
23
```

Binary messages have lower *precedence* than unary messages. Without any grouping notation, the following expression's unary messages will be evaluated first and then passed as arguments to the binary message:

```
Slate> 7 factorial + 3 negated.
5037
Slate> (7 factorial) + (3 negated).
5037
Slate> (7 factorial + 3) negated.
-5043
```

Binary selectors can consist of one or more of the following characters:

```
# $ % ^ * - + = ~ / \ ? < > , ;
```

However, these characters are reserved:

```
@ [ ] ( ) { } . : ! | ` &
```

[1] This removes a source of grammatical complexity in a language where anyone can add new binary selectors or implementations. It is our policy that conventional mathematical notation and visual convenience belong in user interface libraries.

2.2.3 Keyword Messages

A KEYWORD MESSAGE is an alternating sequence of keywords and expressions, generally being a continued infix form. Keywords are identifiers beginning with a letter and ending with a colon. Keyword messages start with the left-most argument along with the longest possible sequence of keyword-value pairs. The SELECTOR of the message is the joining-together of all the keywords into one `Symbol`, which is the *name* of the message. For example,

```
Slate> 5 min: 4 max: 7.
7
```

is a keyword message-send named `min:max:` which has 3 arguments: 5, 4, and 7. However,

```
Slate> 5 min: (4 max: 7).
5
```

is a different kind of expression. *Two* keyword message-sends are made, the first being `max:` sent to 4 and 7, and `min:` sent to 5 and the first result. Note however, that even though the first expression evaluates to the same value as:

```
Slate> (5 min: 4) max: 7.
7
```

that this is still a distinct expression from the first one, with two message-sends of one keyword each instead of one send with two keywords. Actually, this expresses the definition of `min:max:`, although this is perhaps one of the most trivial uses of method names with multiple keywords.

Keywords have the lowest *precedence* of message-sends, so arguments may be the results of unary or binary sends without explicit grouping required. For example, the first expression here is equivalent to the latter implicitly:

```
Slate> 5 + 4 min: 7 factorial max: 8.
9
Slate> (5 + 4) min: (7 factorial) max: 8.
9
```

2.2.4 Implicit-context Messages

Within methods, blocks, and even at the top-level, some expressions may take the surrounding context as the first argument. There is an order for the determination of which object becomes the first argument, which is entirely based on lexical scoping. So, within a block, an implicit send will take the block's run-time context as argument. The next outer contexts follow in sequence, up to the top-level and

what it inherits from, which generally turns out to be the global object that roots the current session.

Specifically, any non-literal expression following a statement-separator or starting an expression within parentheses or other grouping is an implicit-context send.

There are some very common uses of implicit-context sends. In particular, accessing and modifying local variables of a block or method is accomplished entirely this way, as well as returns. For example,

```
[| :i j k |
 j: i factorial.
 k: (j raisedTo: 4).
 j < k ifTrue: [| m |
   j: j - i. m: j. ^ (m raisedTo: 3)].
 k: k - 4.
 k
].
```

is a block which, when invoked, takes one argument and has another two to manipulate. Notice that the local slot `j` is available within the enclosed block that also has a further slot `m`. Local blocks may also *override* the slots of their outer contexts with their input and local slots. In this case, the identifiers `j` and `j:`, for example, are automatically-generated accessing and update methods on the context. Because `j:` is a keyword message, if the assigned value is a keyword message-send result, it must be enclosed in parentheses to distinguish the keyword pattern. The `^ (m raisedTo: 3)` message causes the context to exit prematurely, returning as its value the result of the right-hand argument. All methods have this method defined on them, and it will return out to the nearest named block or to the top-level.

The way this is currently implemented is that the compiler will detect when a local slot is read or modified and it will turn the command into a `loadVariable` or `storeVariable` bytecode. If the a read or write is done to a variable in a parent block, it will be compiled to a `loadFreeVariable` or `storeFreeVariable` which will get the value from the current closure's lexical window array.

2.3 Sequencing Expressions

Statements are the overall expressions between stop-marks, which are periods. In an interactive evaluation context, expressions aren't evaluated until a full (top-level) statement is expressed. The stop mark also means that statement's expression results aren't directly carried forward as an argument to the following expression; side-effects must be used to use the results. More specifically, each expression in the sequence must be evaluated in order, and one expression's side-effects must effectively occur before the next expression begins executing and before any of its side-effects occur.

Slate provides for a bare expression sequence syntax that can be embedded within any grouping parentheses, as follows:

```
Slate> 3 + 4.
7
Slate> (7 factorial. 5
 negated) min: 6.
-5
```

The parentheses are used just as normal grouping, and notably, the `5 negated` expression wraps over a line, but still evaluates that way.[2] If the parentheses are empty, or the last statement in a sequence is followed by a period before ending the sequence, an "empty expression" value is returned, which is `Nil` by convention.

2.4 Methods

METHODS in Slate are basically annotated code blocks (documented in 2.1.1), coupled with annotations of the objects' roles that dispatch to them.

2.4.1 Roles

A relatively unique concept in the language is that objects relate to their methods via an *implicit* idea called a ROLE. A role is an association between an object and a method that applies to it that is similar to a slot, but not directly a slot as it is in Self. Instead of simply being a storage reference, a role associates the object with a *position* in the signature of a method. This is a way of stating that the object "plays a role" in a certain behavior, reinforcing the idea that behavior arrives through cooperation. Furthermore, the behavior is shared among the cooperators, so methods are not "owned" by particular objects; they are not properties.

However, because dispatch can happen on many combinations of arguments, a method name plus the object's position is not sufficient to identify a single method; instead, a group of methods of the same name may be associated via a role with an object at a certain signature position. During method lookup, the appropriate method is found through knowing all of the objects in the signature and knowing which take precedence over others. So a specific role consists of a triple-association between an object, its position in a signature for a method, and the method itself.

When referring to an object's "roles" in general, it usually means the collection of these associations as a whole; what relates the object to the behaviors it participates in.

2.4.2 Method Definitions

Method definition syntax is handled relatively separately from normal precedence and grammar. It essentially revolves around the use of the reserved character "`@`". If any identifier in a message-send argument position is found to contain

[2] We do not consider this expression good style, but it illustrates the nature of the syntax.

the character, the rest of the same send is examined for other instances of the `Symbol`, and the whole send-expression is treated as a template. The parser treats the expression or identifier to the right of the `@` characters as dispatch targets for the method's argument positions; the actual objects returned by the expressions are annotated with a role for their positions.

After the message-send template, there is expected a block expression of some kind, whether a literal or an existing block. Whichever is specified, the parser creates a new block out of it with adjustments so that the identifiers in the dispatching message-send become input slots in the closure. The block should be the final expression encountered before the next stop (a period).

There is a further allowance that an input slot-name specifier may be solely an underscore (but not an underscore followed by anything else), in which case the argument to the method at that position is *not* passed in to the block closure.

This syntax is much simpler to recognize and create than to explain. For example, the following are a series of message definitions adding to boolean control of evaluation:

```
_@True ifTrue: block ifFalse: _ [block do].
_@False ifTrue: _ ifFalse: block [block do].

bool@(Boolean traits) ifTrue: block
"Some sugaring for ifTrue:ifFalse:."
[
 bool ifTrue: block ifFalse: []
].
```

The first two represent good uses of dispatching on a particular individual object (dispatching the ignored argument named "_" to `True` and `False`, respectively) as well as the syntax for disregarding its value. Within their blocks, `block` refers to the named argument to the method. What's hidden is that the block given as the code is re-written to include those arguments as inputs in the header. Also, the objects given to the dispatch annotation are configured to install this method in appropriate roles; because this is hidden, roles are normally an implicit concept and should not concern the user except as an explanation for dispatch.

The latter method appears to have a slightly-different syntax, but this is an illusion: the parentheses are just surrounding a Slate expression which evaluates to an object, much as `True` and `False` evaluate to particular objects; really, any Slate expression can be placed there, assuming that the result of it is what is wanted for dispatch. As a side note, this last method is defined in terms of the first two and is shared, since `True` and `False` both delegate to `Boolean traits` (the object carrying the common behavior of the boolean objects).

2.4.3 Expression-based Definitions

The specialized syntax using the "`@`" special has an equivalent in regular Slate syntax which is often useful for generating new methods dynamically in a non-

ambiguous way. This is a reflective call on the evaluator to compile a method using a certain symbolic name and a sequence of objects that are used for dispatching targets. For example:

```
[| :x :y | Nil] asMethod: #+ on: {True. False}.
```

and

```
_@True + _@False [Nil].
```

are equivalent (while not recommendable) expressions. This raises the question of a place-filler for an argument position which is not dispatched. In that case, Slate provides a unique primitive `NoRole` for this purpose, which provides an analogous role to `Nil`: `NoRole` cannot be dispatched upon. Essentially, this means that the following method definition:

```
c@(Set traits) keyAt: index
[
  c array at: index
].
```

is semantically equivalent to:

```
c@(Set traits) keyAt: index@NoRole
[
  c array at: index
].
```

and furthermore to:

```
[| :c :index | c array at: index] asMethod: #keyAt:
                   on: {Set traits. NoRole}.
```

2.4.4 Lookup Semantics

Message dispatch in Slate is achieved by consulting all of the arguments to that message, and considering what roles they have pertaining to that message name and their position within the message-send. Slate's dispatch semantics are termed "multiple dispatch" to distinguish from "single dispatch" which is typical of most languages based on objects and messages. Whereas most languages designate on object as the receiver of a message, Slate considers all objects involved cooperating participants. During the dispatch process, more than one method can be discovered as a potential candidate. The most *specific* candidate is chosen as soon as its place in the order is determined.

The algorithm achieves a full ordering of arguments: the specificity of the first argument counts more than the second, the second more than the third, and so

on. However, where normal multiple dispatch looks at each argument and uses the most specific supertype to determine specificity (or rather, the most specific parameter type of which the argument is a subtype), Slate instead interprets specificity as DISTANCE in the directed graph of delegations, starting from the particular argument in question as the root.

The DISTANCE notion has the following properties:

- It is determined by a depth-first traversal over the delegate slots, considering delegates at a higher index in the array before lower ones.
- Delegations that lead to cycles are not traversed.
- Repeated finds of a same method do not alter the distance value for it; the first one found is retained.
- The closer (smaller) the DISTANCE of the role to the argument, the more specific it is.

So, Slate's lookup algorithm visits each argument in turn, determining candidate applicable methods as ordered by the DISTANCE notion, and traverses further to look for other possible candidates, unless it rules out the possibility of a more applicable method than a singly-identified one.

The resulting dispatched method satisfies the property that: for any of the arguments, we can find the method on some role reachable by traversing delegations, and that is the closest such method we can find, where former arguments count as being "closer" than any subsequent arguments, and `NoRole` behaves like an "omega distance", as far away as possible.

2.4.5 Optional Keyword Arguments

The mechanism in Slate for specifying optional input arguments uses *optional keywords* for a syntax. They can be added to a method definition, a message-send, or a block header equally. Also, optional keywords can apply to unary, binary, and keyword message syntaxes equally. However, optional arguments cannot affect dispatch, can be provided in any order, and when not provided will start with a `Nil` value, due to their being incidental to the core semantics of the given block or method.

In Method Definitions Method definitions may be annotated with optionals that they support by extending the signature with keyword-localname pairs in arbitrary order as "`&keywordName: argName`". This compiles the method to support `argName` as a local, with optional input.

In Message Sends An optional keyword argument is passed to a method by forming keyword-value pairs as "`&keywordName: someValue`" after the main selector and arguments as normal. Following keywords that have the `&`-prefix will

be collected into the same message-send. A following non-optional keyword will be treated as beginning a new surrounding message-send, but in general, optional keywords raise the precedence of the basis message signature to a keyword level, instead of just unary or binary. Again, the order of the keyword-value pairs is ignored.

In Code Blocks A block can declare optional input keywords in its block header in any order, using "`&argName`" as an input variable declaration, called with the normal convention (when using the block with `do`/`applyTo:`/etc.), whenever the block is invoked with a message-send.

2.4.6 Resending messages or Dispatch-overriding

Because Slate's methods are not centered around any particular argument, the resending of messages is formulated in terms of giving the method activation itself a message. The following are the various primitive protocols involved in resends:

`resend` is the simplest form of resending. It acts on the context to find the next-most-applicable method and invokes it with the exact same set of arguments (including optional parameters passed) as an expression. The result of the `resend` message is the returned result of that method, just like calling any other method.

`methodName findOn: argumentArray` locates and answers the method for the given `Symbol` name and group of argument objects.

`methodName findOn: argumentArray after: aMethod` locates and answers the method following the given one with the same type of arguments as above.

`methodName sendTo: argumentArray` is an explicit application of a method, useful when the `Symbol` name of the method needs to be provided at run-time. It returns the result of the method.

`sendWith:`, `sendWith:with:` and **`sendWith:with:with:`** take one, two, and three arguments respectively as above without creating an array to pass the arguments in. It returns the result of the method.

`methodName sendTo: argumentArray through: dispatchArray` is an extra option to specify a different signature (the `dispatchArray`, where the new lookup starts) for the method than that of the actual argument objects. It returns the result of the method.

Also, both `sendTo:` and `sendTo:through:` accept an `&optionals:` optional keyword which is passed an `Array` of the alternating keyword `Symbol`s (not the names of the locals: those are defined per method) and values to use.

2.4.7 Subjective Dispatch (currently disabled)

The multiple dispatch system has an extended dynamic signature form which can be used to give a "subjective" or "layered" customization of the Slate environment. This is an implementation and slight modification of the Us language features conceived of by the Self authors[Smith 96].

Basic mechanisms

1. Slate dispatch signatures are "enlarged" to support two implicit endpoints: one before the first argument and one after the last argument. We refer to the first role as an "adviser" or `Layer`, and to the second as a `Subject` or "interleaver". The layer role, being "to the left" of the explicit argument positions, has higher precedence than any of them; the subject role has a correspondingly opposite role: it has the lowest precedence of all.
2. Two primitive context-handling methods were added to support invoking code with a different object used for one of these new roles. What happens is that you execute a block `[] seenFrom: someSubject` in order to make all methods defined within dispatched with that object as the subject, and all methods looked up in that context (or any other context "seen from" that object) used with that subject in the dispatch.

The effect of combining these two mechanisms is that there is a means for the user to dynamically (and transparently) extend existing libraries. The `Layer` usage has a more "absolute" power to override, since without dispatching on any other arguments, a method defined in a layer will match a message before any other can. The `Subject` usage has a more fine-tuned (or weaker, in another sense) ability to override, since without any other dispatching, a method defined with a certain subject will never be called. However, taking an existing method's signature and defining a customized version with a subject will allow customizing that specific method without affecting any other method with that selector.

Important features

- Methods defined with a special subject or layer *persist* with those objects, since they are just dispatch participants accessible via roles.
- Resending messages works just the same within subjective methods as in normal methods; the same dispatch mechanism is in effect, so the ability to combine or extend functionality is available.
- Nesting subjective scopes has a *dynamic scoping* effect: the actions taken within have run-time scope instead of corresponding exactly to how code is lexically defined. This gives the compositional effect that should be apparent when viewing nested subjective scopes.
- Methods defined in non-subjective contexts have no subject or layer rather than any "default" subject: they are for most purposes, "objective".

The core elements

`Subject` the type of object which provides an appropriate handle for subjective interleaving behavior in dynamically overriding or extending other methods' behaviors.

`Layer` the type of object which provides an appropriate handle for subjective layering behavior in dynamically overriding or extending other methods' behaviors.

`[] seenFrom: aSubject` executes the contents of the block with the given `Subject` dynamically affecting the execution of the expressions.

`aLayer layering: []` executes the contents of the block with the given `Layer` dynamically affecting the execution of the expressions.

`[] withoutSubject` executes the contents of the block without any subject.

`[] withoutLayers` executes the contents of the block without any layer.

2.5 Type Annotations

Input and local slots' types can be specified statically for performance or documentation reasons, if desired. The special character "`!`" is used in the same manner as the dispatch annotation "`@`", but type-annotations can occur anywhere. The type system and inference system in Slate is part of the standard library, and so is explained later in 3.14 on page 55.

2.6 Macro Message-sends

In order to manipulate syntax trees or provide annotations on source code, Slate provides another form of message-send called a *macro-level* message send. Sends of this sort have as their arguments the objects built for the expressions' shapes. Furthermore, the results of evaluation of macro-sends are placed in the syntax tree in the same location as the macro-send occupied.

Preceding any selector with a back-tick (`) will cause it to be sent as a macro. This means that the message sent will be dispatched on Slate's `Syntax Node` objects, which are produced by the parser and consumed by the compiler. Macros fit into the process of compiling in this order: the text is processed by the `Lexer` into a stream of tokens, which are consumed by the `Parser` to produce a (tree-)stream of `Syntax Node`s. Before being passed to the compiler, the `macroexpand` method is recursively called on these syntax trees, which invokes every macro-level message-send and performs the mechanics of replacing the macro-send with the result of the method invoked. With this in mind, the Slate macro system is offered as a flexible communication system between the code-writer and the compiler or other tools or even other users, using parse trees as the medium.

As an example of the expressiveness of this system, we can express the type annotation and comment features of the Slate language in terms of macros:

- Type-annotation via "`expression!type`" could be replaced by "`someExpression 'type: assertedTypeExpression`" where the `'type:` macro simply sets the `type` slot for the expression object.
- Comments could be applied specifically to particular expressions. For example, following a syntax element with a comment could be implemented by "`theExpression 'comment: commentString`", wrapping the syntax node with a comment annotation.
- Compile-time evaluation of any expression can be accomplished by calling `'evaluate` on it. This also subsumes the `#`-prefix for array literals and expression sequences which accomplishes that for those syntax forms.

2.6.1 Defining new Macro-methods

Macros must be dispatched (if at all) upon the traits of expressions' syntactic representation. This introduces a few difficulties, in that some familiarity is needed with the parse node types in order to name them. However, only two things need to be remembered:

1. The generic syntax node type is `Syntax Node traits`, and this is usually all that is necessary for basic macro-methods.
2. Syntax node types of various objects and specific expression types can be had by simply quoting them and asking for their traits, although this might be too specific in some cases. For example, `4 'quote traits` is suitable for dispatching on Integers, but not Numbers in general, or `(3 + 4) 'quote traits` will help dispatch on binary message-sends, but not all message-sends. Luckily, `[] 'quote traits` works for blocks as well as methods.

2.6.2 Quoting and Unquoting

A fundamental application of the macro message-send system is the ability to obtain syntax trees for any expression at run-time. The most basic methods for this are `'quote`, which causes the surrounding expression to use its quoted value as the input for even normal methods, and `'unquote` results in an inversion of the action of `'quote`, so it can only be provided within quoted expressions.

An abbreviated form of quotation may be used by surrounding an expression with `'()` which has the same effect as sending `'quote`. The parentheses may be omitted if the inner expression cannot be interpreted as a message-send (applicable to literal blocks, arrays, numbers, and so on).[3]

[3] Lisp macro system users will note that this effectively makes `'quote` the same as quasi-quotation.

Labelled Quotation In experience with Lisp macros, nested quotation is often found necessary. In order to adequately control this, often the quotation prefix symbols have to be combined in non-intuitive ways to produce the correct code. Slate includes, as an alternative, two operations which set a label on a quotation and can unquote within that to the original quotation by means of referencing the label.

Most users need time to develop the understanding of the need for higher-order macros, and this relates to users who employ them. For reference, a Lisp book which covers the subject of higher-order macros better than any other is *On Lisp*[Graham 94]. However, it's also been said that Lisp's notation and the conceptual overhead required to manage the notation in higher-order macros keeps programmers from entering the field, so perhaps this new notation will help.

The operators are `expr1 `quote: aLiteral` and `expr2 `unquote: aLiteral`, and in order for this to work syntactically, the labels must be equal in value and must be literals. As well, the unquoting expression has to be a sub-expression of the quotation. The effect is that nesting an expression more deeply does not require altering the quotation operators to compensate, and it does indicate better what the unquoting is intended to do.

2.6.3 Message Cascading

Many object-oriented code idioms involve repeated message-sends to the same object. This is common in creating and setting up a new object, or activating many behaviors in sequence on the same thing (used in a lot of UI code - or more generally with objects that have many variables or are "facade" objects). Take notice of the period after the last message in the block. If you forget, you'll return the last result instead of the original object your cascading into.

Smalltalk-80 included a syntax feature to elide the first argument when repeating message-sends. In Slate, the first argument of a message send is not a special "receiver", so support in the basic syntax does not make sense. The solution is a macro-method which takes an expression (well, its result) and a block, and enhances the block so that the result of the expression becomes the implicit context for statement-level (top-level) message expressions. For example:

```
Slate> (addPrototype: #Something derivedFrom: {Cloneable})
  `>> [addSlot: #foo. addSlot: #bar. ].
("Something traitsWindow" bar: Nil. foo: Nil)
```

In this expression, a prototype is created and returned by the first expression, and the block is evaluated with that object substituted for the context, so that the `addSlot:` calls apply to it directly. We recommend you don't define things this way. Instead use `namespace define: #Something &parents: {Cloneable} &slots: {#foo -> Nil. #bar -> Nil }.`

There is an additional possibility that it takes care of by allowing the user to specify an input variable to the block, which will also allow the code within to refer

to the object explicitly. Also, the default return value for empty-last-statements is modified to be this object instead of the usual `Nil`. Without such features, a method which returns its only argument is needed; in Smalltalk-80, the `yourself` method performs this role at the end of a cascade. To see the effect this would have on Slate library code, take a collection creation method as an example:

```
set@(Set traits) new &capacity: n
[| newSet |
  newSet: set clone.
  newSet contents: (set contents new &capacity: ((n ifNil: [0]) max: 1)).
  newSet tally: 0.
  newSet
].
```

could become:

```
set@(Set traits) new &capacity: n
[set clone `>>
  [contents: (set contents new &capacity: ((n ifNil: [0]) max: 1)).
  tally: 0. ]
].
```

So in this new method, an object is made by cloning the argument `Set`, and then the contents and tally are assigned as before, but not needing to refer to the object explicitly, and then returning the object (if the ending period is left off, it'll return the value of the last explicit expression, just as normal blocks). However, the message-sends which are not leading a statement do not (and cannot) use the object as implicit context. This is intentional, as the results of this kind of pervasive change are more drastic and would be an entirely different kind of idiom. If a reference to the implicit context argument is needed in the block, an input argument may be specified in the block header, and the macro method will bind it to that object. A notable property of the `` `>> `` method, since it returns resulting values and has a binary selector, is composability in a data-flow manner.

2.6.4 Slots as Block Variables

Another macro allows for the slots of an object to appear as local variables in a method, responding both to accessors and mutators in the appropriate way (by modifying the object's slots).

So, the following expression names some slots in the first argument which should be available as inputs to the second argument, a block. This is analogous to having Smalltalk's direct slot reference syntax (or Self's, for that matter), or to Lisp's with-slots macro. (Again, you shouldn't generally shouldn't create objects this way.)

```
Slate> Cloneable clone
  `>> [addSlot: #x valued: 2. addSlot: #y valued: 2]
  `withSlotsDo: [| :x :y | x + y].
4
```

2.6.5 Expression Substitution (Not Yet Implemented)

`` `with:as: `` is a proposed protocol for transparent substitution of temporary or locally-provided proxies for environment values and other system elements. This should provide an effective correspondent of the functionality of Lisp's "`with-`" style macros.

2.6.6 Source Pattern-matching (Not Yet Implemented)

A future framework for expansion will involve accommodating the types of source-level pattern-matching used in tools for manipulating code for development, as in the Smalltalk Refactoring Browser.

2.7 Literal Syntax

2.7.1 Numbers

Integers Integers are read in as an arbitrary-precision sequence of digits, without separators.

Floats Floats are read in as an arbitrary-precision sequence of digits, with a period noting the decimal position.

Radix Prefixes Integers or Floats may be entered with radix up to 36, using the digits 0 to 9 and then A to Z in order, by prefixing the literal with the radix (numeric base) and '`r`'. So `3r100` evaluates to 9, and `2r0.1` evaluates to `0.5`. Case is disregarded for the extra-decimal digits. The Float radix notation is always read in as a Float, and not an infinite-precision Fraction.

2.7.2 Characters

Slate's default support for character literals uses the `$` symbol as a prefix. For example, `$a`, `$3`, `$>`, and `$$` are all Character object literals for `a`, `3`, `>`, and `$`, respectively. Printable and non-printable characters require backslash escapes as shown and listed in Table 1 on the next page.

Character name	Literal
Escape	`$\e`
Newline	`$\n`
Carriage Return	`$\r`
Tab	`$\t`
Backspace	`$\b`
Null	`$\0`
Bell	`$\a`
Form Feed	`$\f`
Vertical Feed	`$\v`
Space	`$\s`
Backslash	`$\\`

Table 1: Character Literal Escapes

2.7.3 Strings

Strings are comprised of any sequence of characters surrounded by single-quote characters. Strings can include the commenting character (double-quotes) without an escape. Embedded single-quotes can be provided by using the backslash character to escape them (`\'`). Slate's character literal syntax also embeds into string literals, omitting the `$` prefix. All characters that require escapes in character literal syntax also require escapes when used within string literals, with the exception of double-quote marks and the addition of single-quote marks.

The following are all illustrative examples of Strings in Slate:

```
'a string comprises any sequence of characters, surrounded by single quotes'
'strings can include the "comment delimiting" character'
'and strings can include embedded single quote characters by escaping\' them'
'strings can contain embedded
newline characters'
'and escaped \ncharacters'
'' "and don't forget the empty string"
```

2.7.4 Symbols

`Symbol` literal syntax starts with the pound sign character (`#`) and consists of all following characters up to the next non-escaped whitespace or reserved character (whichever comes first), unless the pound sign is followed exactly by a string literal (in single quotes), in which case the string's contents become the identifier for the `Symbol`. So, for example, the following are all valid `Symbol`s and `Symbol` literals:

```
#$
#20
```

```
#+
#key:word:expression:
#something_with_underscores
#'A full string with a \nnewline in it.'
#'@'
```

A property of `Symbols` and their literals is that any literal with the same value as another also refers to the *same instance* as any other `Symbol` literal with that value in a Slate system. This allows fast hashes and comparisons by identity rather than value hashes. In particular, as with Slate identifiers, a `Symbol`'s value is case-sensitive, so `#a` and `#A` are distinct.

Internally, Slate currently keeps one global table for `Symbol`s, and uses individual context objects to hold local bindings.[4]

2.7.5 Arrays

Arrays can be literally and recursively specified by curly-brace notation using stops as separators. Array indices in Slate are 0-based. So:

```
{4. 5. {foo. bar}}.
```

returns an array with 4 in position 0, 5 at 1, and an array with objects `foo` and `bar` inserted into it at position 2.

Immediate array syntax - `#{4. 5. {foo. bar}}` - is provided as an alternative to create the array when the method is compiled, instead of creating a new array on each method invocation. The syntax is identical except that the first opening brace is preceded by the pound sign. The disadvantage is that no run-time values will be usable.

A special "literal array" syntax is also provided, in the manner of Smalltalk-80, in which all tokens within are treated symbolically, evaluating to an array of literals as read (but not evaluated) by Slate. Naturally, these are all evaluated when the surrounding context is compiled. For example:

```
Slate> #(1 2 3).
{1. 2. 3}
Slate> #(3 + 4).
{3. #'+'. 4}
Slate> #(quux: a :bar).
{#quux:. #a. #:bar}
Slate> #(1 . _ 2e3).
{1. #'.'. #_. 2000.0}
```

[4] Future, bootstrapped releases may provide for partitioning of the global table.

2.7.6 Blocks

Block syntax basics were covered in 2.1.1; the precise, full specification includes more features and outlines some necessary logical rules. Primarily, blocks are square-bracket-delimited statement sequences with an optional header that specifies input and local slots (input slots being *arguments*).

Slot names must be valid unary message selectors (see 2.2.1). Inputs are distinguished by a prefix colon character (`:`), and must occur in the same positional order that the invocation will use or expect, although they can be interspersed among other slot declarations at will.

Optional keyword arguments are specified with an ampersand prefix character (`&`), and may occur in any order.

For example,

```
[| x :y &z :w | ]
```

evaluates to a block which takes inputs `y` and `w` in that order, has locals `x` (and `z`), and takes an optional parameter to specify `z`'s value when called.

A single "rest" parameter which becomes an array containing all extra positional (non-keyword) arguments passed may be specified once in the header in any position relative to the other parameters, prefixed with an asterisk (e.g. `*rest`).

Blocks may be used to perform arbitrary compile-time calculations, using the `#`-prefix as used for literal arrays and strings. So `#[3 + 4]` will result in `7` in the resulting code for the surrounding context (perhaps a method or top-level expression), as though the block were never there.

3 The Slate World

3.1 Overall Organization

3.1.1 The lobby

The `lobby` is the root namespace object for the Slate object system; it is the "room" by which objects enter the Slate world. All "global" objects are really only globally accessible because the lobby is delegated to by lexical contexts, directly or indirectly. The lobby in turn may (and often does) delegate to other namespaces which contain different categorized objects of interest to the applications programmer, and this can be altered at run-time.

Every object reference which is not local to a block closure is sent to the enclosing namespace for resolution, which by default is the root namespace, the `lobby` (nested closures refer first to their surrounding closure). The `lobby` contains a loopback slot referring to itself by that name. To add or arrange globals, either implicit sends or explicit references to the lobby can be made. (Consider it good style to directly reference it.)

The lobby is essentially a threading context, and in the future bootstrap will be instantiable in that sense.

3.1.2 Naming and Paths

The `lobby` provides access to the major `Namespaces`, which are objects suitable for organizing things (for now, they are essentially just `Cloneable` objects). The most important one is `prototypes`, which contains the major kinds of shared behavior used by the system. Objects there may be cloned and used directly, but they should not themselves be manipulated without some design effort, since these are global resources, having a name-path identifier which can be freely shared. `prototypes` is inherited by the lobby, so it is not necessary to use the namespace path to identify, for example, `Collection` or `Boolean`. However, without explicitly mentioning the path, adding slots will use the lobby or the local context by default. To use the current namespace, an implicit-context message `here` is provided which will answer the nearest surrounding namespace object in the context, which is the `lobby` by default, but is polymorphic to namespace shifts.

The `prototypes` namespace further contains inherited namespaces for, by example, collections, and can be otherwise enhanced to divide up the system into manageable pieces.

3.2 Core Behaviors

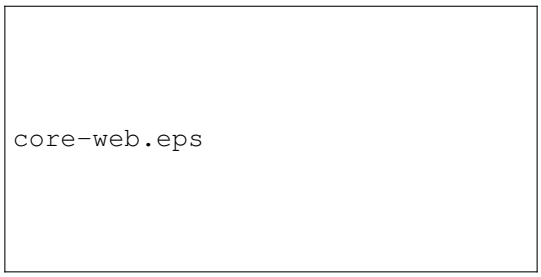

Figure 1: Core Object Inheritance

Slate defines several subtle variations on the core behavior of objects:

`Root` The "root" object, upon which all the very basic methods of slot manipulation are defined.

`Oddball` The branch of `Root` representing non-cloneable objects. These include built-in 'constants' such as the `Boolean`s, as well as literals (value-objects) such as `Character`s and `Symbol`s. Note that `Oddball` itself defines a `clone` method, but that method will only work once, in that you can clone `Oddball` but not objects made by cloning `Oddball`.

Nil `Nil` is an `Oddball` representing "no-object".

NoRole `NoRole` is an `Oddball` representing a non-dispatching participant in a method definition. Methods cannot be defined directly on `NoRole`.

Derivable `Derivable` objects respond to `derive`, which means they can be readily used for extension.

Cloneable A `Derivable` that can be `clone`d.

Method A `Cloneable` object with attributes for supporting execution of blocks (with closure semantics, notably) and holding compiled code and its attributes.

3.2.1 Default Object Features

Identity == returns whether the two arguments are identical, i.e. the same object, and ~== is its negation. Value-equality (= and its negation ~=) defaults to this.

Printing `printString` returns a printed (string) representation of the object. `printOn:` places the result of printing onto a designated `Stream` (`print` will invoke `printOn:` on the `Console`). This should be overridden for clarity.

Delegation-testing `isReally:` returns whether the first object has the second (or its traits object if it is not a `Trait`) as one of its delegated objects, directly or indirectly.

Kind-testing `is:` returns whether the first object has the same kind as the second object, or some derived kind from the second object's kind. By default, `is:` is `isReally:`; overrides can allow the user to adapt or abuse this notion where delegation isn't appropriate but kind-similarity still should hold. `isSameAs:` answers whether the arguments have the same traits object.

Hashing A quick way to sort by object value that makes searching collections faster is the `hash` method, which by default hashes on the object's identity (available separately as `identityHash`), essentially by its birth address in memory. More importantly, this is how value-equality is established for collections; if an object type overrides =, it must also override the `hash` method's algorithm so that `a = b` ⇒ `a hash = b hash`.

Cloning The `clone` method is fundamental for Slate objects. It creates and returns a new object identical in slot names and values to the argument object, but with a new unique identity. As such, it has a very specific meaning and should only be used that way.

Copying The `copy` method makes a value-equal (=) object from the argument and returns the new object. This should be overridden as necessary where = is overridden. The default case is to clone the original object.

Conversion/coercion the `as:` protocol provides default conversion methods between types of objects in Slate. Some primitive types, such as `Number`, override this. The `as:` method has a default implementation on root objects: if no converter is found or if the objects are not of the same type, the failure will raise a condition. Precisely, the behavior of `a as: b` is to produce an object based on `a` which is as much like `b` as possible.

3.2.2 Oddballs

There are various `Oddball`s in the system, and they are non-cloneable in general. However, `Oddball` itself may be cloned, for extension purposes.

3.3 Introspection

3.3.1 Slot Queries

Objects may be queried directly for various attributes that they have. These should not be used in ordinary "non-private" code, only in specific uses which need to operate on arbitrary object-slot structures or inside of accessor methods.

`atSlotNamed:` answers the value of the slot with the given name.

`atSlotNamed:put:` fills the delegate slot with the given name. It answers the object modified.

`slotNames` answers an `Array` of the names of the object's direct slots.

`_delegates` answers an `Array` of the names of the object's direct delegate slots.

`slotsDo:` applies a block to each slot's value.

`delegatesDo:` applies a block to each delegate slot's value.

`addAccessorFor:` defines a default accessor (aka "getter") method for the slot with the given name.

`addMutatorFor:` defines a default mutator (aka "setter") method for the slot with the given name.

`addAccessor:for:` installs the given method as accessor for the slot with the given name.

`addMutator:for:` installs the given method as mutator for the slot with the given name.

3.3.2 Method Queries

Slate includes a protocol for common query operations on `Method` objects:

`selector` answers the `Symbol` naming the `Method` if it has been defined and installed under some objects' roles, or `Nil` if it is unnamed (an ordinary block).

`isNamed` answers whether the `Method` is a named method; that is, whether it is installed with a dispatch signature and selector.

`arity` answers the number of arguments expected as input to the `Method`. This excludes any optional keyword arguments or rest arguments that it may accept. This also applies to the quoted form of a literal method.

`acceptsAdditionalArguments` answers whether the `Method` takes a rest parameter.

`allSelectorsSent` answers an `Array` of all `Symbol`s that the block directly (in its defining source) sends. This also applies to the quoted form of a literal method.

`methodsNamed:` answers all distinct `Method` objects with the given name stored in roles on the given object.

`methodsAt:` answers all distinct `Method` objects stored in the role position given on the given object.

`methodsNamed:at:` answers all distinct `Method` objects stored in roles on the given object with the given name and at the given role position.

`methodsSending:` answers all distinct `Method` objects stored in roles on the given object which send the given selector.

`signature &in:` answers a `Signature` object (roles plus selector) corresponding to the given (named) `Method`'s definition. This currently involves a search and takes a variable amount of time.

`Symbol implementations` answers a `Set` of all `Method`s whose `selector` is the argument. An `&in:` optional parameter restricts the search to one object (as namespace).

`Symbol senders` answers a `Set` of all `Method`s which send the argument selector. An `&in:` optional parameter restricts the search to one object (as namespace).

`Symbol macroSenders` answers a `Set` of all `Method`s which send the argument selector. An `&in:` optional parameter restricts the search to one object (as namespace).

3.4 Traits

Slate objects, from the root objects down, all respond to the message `traits`, which is conceptually shared behavior but is not as binding as a class is. It returns an object which is, by convention, the location to place shared behavior. Most Slate method definitions are defined upon some object's `traits` object. This is significant because cloning an object with a traits delegation slot will result in a new object with the same object delegated-to, so all methods defined on that traits object apply to the new clone. There is one core method which drives derivation:

`myObject derive &mixins: &rejects:` will return a new clone of the `Derivable` object with a traits object which is cloned from the original's traits object, and an immutable delegation slot set between the traits objects. If mixins are given, it will include more immutable delegation links between the new traits and the traits of the array's objects, in the given order, which achieves a structured, shared behavior of static multiple delegation. Note that the delegation link addition order makes the right-most delegation target override the former ones in that order. One interesting property of this method is that the elements of the mixins do not have to be `Derivable`.

In practice, the following wrapper is the appropriate method to use in common situations, since it handles the installation of the prototype in a cleaner manner:

`obj addPrototype: name derivedFrom: parentsArray` will perform the effects of `derive` using all the elements of the `Sequence` in the same order as `derive`. It also assigns the name to the traits object's name attribute (which should be a `Symbol`) as well as using the name for the attribute between the surrounding object and the new prototype. Finally, it will compare the delegation pattern of the new object with the old, and only replace the old if they differ. In either case, the installed object is what is returned, and the existing traits object will be re-used if there is one.

`obj define: name &parents: parentsArray &slots: slotSpecs` performs the effects of `derive` using all the elements of the `Sequence` in the same order as `derive`. The default parent is just `Cloneable`. It also assigns the name to the traits object's name attribute (which should be a `Symbol`) as well as using the name for the attribute between the surrounding object and the new prototype. The `slotSpecs` needs to be an array with either `Symbol`s or `Association`s from `Symbol`s to values, to specify (mutable) state-slots and their attributes. Finally, it will compare the delegation pattern of the new object with the old, and only replace the old if they differ. In either case, the installed object is what is returned, and the existing traits object will be re-used if there is one. A `define:&builder:` form is also provided which takes a block and uses its resulting value for the object; `&builder:` can be used in conjuction with `&slots:`, but not with `&parents:`, as no derivation is relevant.

As with any method in Slate, these may be overridden to provide additional automation and safety in line with their semantics.

Traits Windows Beneath this simple system is a level of indirection which allows for greater control *when necessary* over the combination of parent behaviors; for most purposes, they may be ignored. What is actually the case is that each object points to an object in a `traitsWindow` delegation slot. `traitsWindow` is defined to be the first delegate in an object's delegates array and `traits` is defined to be the last delegate in an object's `traitsWindow`.

Because the delegates of an object are easily changed, there is a great deal of control over the order of precedence This means that nearly all of the problems of precedence in a multiple inheritance graph may be resolved flexibly.[5] There is also a hint to the lookup procedure in the `traitsWindow` objects that a meta-level has been reached. Once one meta-transition has occurred during lookup, the algorithm will not traverse such a transition again; thus Slate avoids a lookup confusion between methods meant for regular objects and those meant for the traits themselves.

3.5 Blocks, Booleans, and Control-Flow

3.5.1 Boolean Logic

Slate's evaluator primitively provides the objects `True` and `False`, which are clones of `Boolean`, and delegate to `Boolean traits`. Logical methods are defined on these in a very minimalistic way. Table 2 shows the non-lazy logical methods and their meanings.

Description	Selector
AND/Conjunction	`/\`
OR/Disjunction	`\/`
NOT/Negation	`not`
EQV/Equivalence	`eqv:`
XOR/Exclusive-OR	`xor:`

Table 2: Basic Logical Operators

3.5.2 Basic Conditional Evaluation

Logical methods are provided which take a block as their second argument (`/\`, `\/`, `and:`, `or:`, `xor:`, `eqv:`). By accepting a block as the second argument, they can and do provide conditional evaluation of the second argument only in the case that the first does not decide the total result automatically[6]. Blocks that evaluate logical expressions can be used lazily in these logical expressions. For example,

[5]The classical illustration of multiple inheritance issues is the "diamond" problem where one parent is inherited via two different paths indirectly. Solutions that involve choosing inheritance graph traversal orders for the whole language are insufficiently flexible in dealing with multiple inheritance confusion, since various protocols need to interact independently.

[6]However, support for blocks in the second argument position may be incorporated into the non-lazy selectors as different methods in the future, making some of these obsolete.

```
(x < 3) /\ [y > 7].
```

only evaluates the right-hand block argument if the first argument turns out to be `True`.

```
(x < 3) \/ [y > 7].
```

only evaluates the right-hand block argument if the first argument turns out to be `False`.

In general, the basic of booleans to switch between code alternatives is to use `ifTrue:`, `ifFalse:`, and `ifTrue:ifFalse:` for the various combinations of binary branches. For example,

```
x isNegative ifTrue: [x: x negated].
```

ensures that `x` is positive by optionally executing code to assign a positive form if it's not. Of course if only the result is desired, instead of just the side-effect, the entire expression's result will be the result of the executed block, so that it can be embedded in further expressions.

Conditional evaluation can also be driven by whether or not a slot has been initialized, or whether a method returns `Nil`. There are a few options for conditionalizing on `Nil`:

`expr ifNil: block` and **`expr ifNotNil: block`** execute their blocks based on whether the expression evaluates to Nil, and returns the result.

`expr ifNil: nilBlock ifNotNil: otherBlock` provides both options in one expression.

`expr ifNotNilDo: block` applies the block to the expression's result if it turns out to be non-`Nil`, so the block given must accept one argument. **`ifNil:ifNotNilDo:`** is also provided for completeness.

There is also a "case-switch" style idiom:

`expr caseOf: cases otherwise: defaultBlock` takes the result of the first argument and the `cases` array of `Association`s (made with `->`) from objects to blocks and executes the first block associated with a value equal to the expression. If none match, the last argument is executed; the otherwise: clause may be omitted, to just do / return nothing.

3.5.3 Early Returns

Control-flow within methods can be skipped with a return value using the `^` message to the context. `^` takes its second argument and exits the nearest surrounding lexical scope that is a named method (and not an anonymous code block) with that value. This message is a real binary message with no special precedence, so disambiguation is often needed. For example,

```
^ 4
^ n factorial
```

return the expressions on the left as a whole, but

```
^ 3 + 4
^ (3 + 4)
^ set collect: [| :each | each name]
^ (set collect: [| :each | each name])
```

represent expression variations where a lack of disambiguation results in returning an unintended answer.

3.5.4 Looping

Slate includes various idioms for constructing basic loops. These are all built from ordinary methods which work with code blocks, and any ordinary code may define similar idioms rather quickly.

`loop` executes the block repeatedly, indefinitely[7].

`n timesRepeat: block` executes the block `n` times.

`condition whileTrue: block` and **`condition whileFalse: block`** execute their blocks repeatedly, checking the condition before each iteration (implying that the body block will not be executed if the condition is initially false).

`whileTrue` and **`whileFalse`** execute their blocks repeatedly, checking the return value before repeating iterations (so that they will always execute at least one time).

`a upTo: b do: block` and **`b downTo: a do: block`** executes the block with each number in turn from `a` to `b`, inclusive.

`upTo:by:do:` and **`downTo:by:do:`** executes the block with each number in turn in the inclusive range, with the given stepping increment.

`a below: b do: block` and **`b above: a do: block`** act identically to the previous method except that they stop just before the last value. This assists in iterating over array ranges, where the 0-based indexing makes a difference in range addresses by one, avoiding excessive use of `size - 1` calls.

`below:by:do:` and **`above:by:do:`** vary on the previous methods with the given stepping increment.

[7] This is currently implemented through a compiler rule.

3.5.5 Method Operations

Slate provides some powerful methods for operating on `Method` (block) objects themselves. The answers of these operations are also fully-fledged `Method` objects which respond to all of the normal queries and are useful anywhere that a `Method` is accepted.

`**` composes two `Method`s in a functional manner, so that $(F \circ G)(x) = F(G(x))$; basically the result of applying the first to the result of the second (which in turn consumes the input). This operation is associative, so grouping of continued applications should not be a concern. This semantics assumes that the inner/right function G may take any number of arguments, and that F takes 1. If both of these are false, then there is an extended form of the semantics where the arity of F is used to group the results of applying G to each element in the argument array. The groups are then passed as a whole to one invocation of F.

**`er** takes a literal `Symbol` taken as a message selector and expands into an appropriate `Method` which takes an appropriate number of arguments for the selector and sends the selector to those arguments, answering the result. E.g. `#+`er` expands into `[| :x :y | x + y]` and `#name`er` expands into `[| :x | x name]`.

converse takes a `Method` and answers a new `Method` with the same body but with input argument order reversed.

<- takes a `Method` and an object and returns a new `Method` with the same body but with the object substituted for the first argument, so it takes one less argument. This is also known as *currying*. `<-1`, `<-2`, `<-3`, and `<-*` curry the next three argument positions and the last argument, respectively. `fill:with:` fills the N'th argument.

**`commutatively** takes a literal `MethodDefinition` and defines it with two signatures, one reversed from the original definition, with the same body.

3.6 Symbols

`Symbol`s in Slate are basically a large group of `String`s sorted by identity into a global table `Symbols`. A new `Symbol` is created or identified by taking a `String` and calling the `intern` method on it (or alternately, the context message `intern:`). Repeatedly `intern`'ing a same `String` value or `Symbol` object will return the exact same `Symbol` object. In order to perform `String` operations (see the `Sequence` and `String` sections3.8.2), the `Symbol` must be asked for its `name` which gives a separate `String` with its value; the results of these operations will still be `String`s and again must be `intern`'ed to create a new `Symbol`.

3.7 Magnitudes and Numbers

3.7.1 Basic Types

Magnitude the abstract protocol for linearly-comparable objects, following <, >, <=, >=, and =.

Number the abstract type of dimensionless quantities.

Integer integral quantities, generally.

SmallInteger machine-word-limited integer values (minus 1 bit for the immediate-value flag). Their normal protocol will not produce errors inconsistent with mathematic behavior of `Integer`s, however: instead of overflows, `BigInteger` objects of the appropriate value are returned.

BigInteger larger `Integer`s, implemented by wrapping `ByteArray`s with the appropriate behavior.

Fraction An exact representation of a quotient, or rational number.

Float A low-level floating-point numeric representation, being inexact. Floats are currently only implemented as `SingleFloat`, a single-precision floating-point number representation.

Complex A complex number, similar to a pair of real numbers.

3.7.2 Basic Operations

All of the normal arithmetic operations (i.e. +, −, *, /) are supported primitively between elements of the same type. Type coercion has to be done entirely in code; no implicit coercions are performed by the virtual machine. However, the standard library includes methods which perform this coercion. The interpreter also transparently provides unlimited-size integers, although the bootstrapped system may not do so implicitly.

The following are the rest of the primitive operations, given with an indication of their "signatures":

Float raisedTo: Float is simple floating-point exponentiation.

Integer as: Float extends an integer into a float.

Float as: Integer truncates a float.

Integer bitOr: Integer performs bitwise logical OR.

Integer bitXor: Integer performs bitwise logical XOR.

Integer bitAnd: Integer performs bitwise logical AND.

Integer bitShift: Integer performs bitwise logical right-shift (left-shift if negative).

Integer bitNot performs bitwise logical NOT.

Integer >> Integer performs logical right-shift.

Integer << Integer performs logical left-shift.

Integer quo: Integer returns a quotient (integer division).

Many more useful methods are defined, such as `mod:`, `reciprocal`, `min:`, `max:`, `between:and:`, `lcm:`, and `gcd:`. Slate also works with `Fractions` when dividing Integers, keeping them lazily reduced.

3.7.3 Non-core Operations

zero The zero element for the type of number.

isZero Whether the number is the zero element for its type.

isPositive/isNegative Whether it's positive or negative.

abs The absolute value of the number.

sign The sign of the number.

negated Returns *-x* for *x*.

gcd: Greatest common divisor.

lcm: Least common multiple.

factorial Factorial.

mod:/rem:/quo: Modulo division, remainder, and quotient.

reciprocal Constructs a new fraction reciprocal.

min: The lesser of the arguments. The least in cases of `min:min:`.

max: The greater of the arguments. The greatest in cases of `max:max:`.

a between: b and: c Whether `a` is between `b` and `c`.

truncated/fractionPart answers the greatest integer less than the number, and the corresponding difference as a fraction (or a float for `Float`).

reduced Only defined on `Fraction`, this is the lazily-applied reducer; it will be invoked automatically for arithmetic operations as necessary, but is useful when only the reduced form is needed.

readFrom: This takes a `String` or a `Stream` with compatible contents and parses the first available data as the type of the argument. If the return value is not the same as the argument type, an error is signalled (beyond any normal parsing errors). If the value is valid, it is returned; otherwise only `Nil` will be available.

3.7.4 Limit Numerics

PositiveInfinity is greater than any other `Magnitude`.

NegativeInfinity is lesser than any other `Magnitude`.

LargeUnbounded A `Magnitude` designed to represent non-infinite, but non-bounded ("as large as you like") quantities.

PositiveEpsilon is as small as you like, but positive and greater than zero.

NegativeEpsilon is as small as you like, but negative and lesser than zero.

3.7.5 Dimensioned Units

There is an entire system for handling dimensioned units and their various combinations and mathematical operations. There is included support for SI units, and common English units; furthermore, any object may conceivably be used as a base unit. See the '`src/lib/dimensioned.slate`' file for an overview.

3.8 Collections

Slate's collection hierarchy makes use of composing multiple behaviors (via inheritance) to provide a collection system that can be reasoned about with greater certainty, and that can be extended more easily than other object-oriented languages' collection types. Primarily Slate collections are mutable, which means that basic modifications occur destructively on the collection object.

Figure 2 shows the overview of the collection types, and how their inheritance is patterned.

All collections support a minimal set of methods, including support for basic internal iteration and testing. The following are representative core methods, and are by no means the limit of collection features:

Testing Methods

isEmpty answers whether or not the collection has any elements in it.

includes: answers whether the collection contains the object.

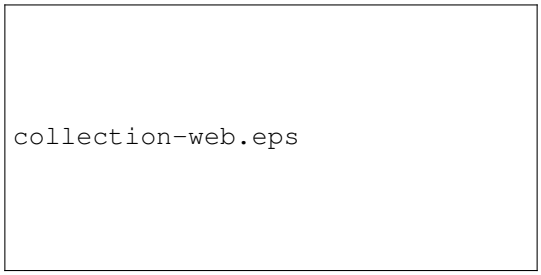

Figure 2: Core Collections Inheritance

Properties

`size` answers the number of elements in it. This is often calculated dynamically for extensible collections, so it's often useful to cache it in a calling method.

`capacity` answers the size that the collection's implementation is currently ready for.

Making new collections

`new &capacity:` answers a new (empty) collection of the same type that is sized to the optional argument or a sensible default per type.

`newSizeOf:` answers a new collection of the same type that is sized to same size that the argument has.

`as:` ***via*** **`newWithAll:`** has extensive support in the collection types to produce collections of the type of the second with the contents of the first collection (*vice versa* for `newWithAll:` arguments).

Iterating

`do:` executes a block with `:each` (the idiomatic input slot for iterating) of the collection's elements in turn. It answers the original collection.

`collect:` also takes a block, but answers a collection with all the results of those block-applications put into a new collection of the appropriate type.

`select:` takes a block that answers a `Boolean` and answers a new collection of the elements that the block filters (answers `True`).

`reject:` performs the logical opposite of `select:`, answering elements for which the block answers `False`.

inject: init into: accumulator takes a two-argument accumulation block and applies it across the collection's elements. The initial value given becomes the first argument, and is replaced through the iterations with the result of the block.

reduce: takes a two-argument block and performs the same action as `inject:into:` only using one of the collection's elements as an initial value.

project: takes a block, answering a `Dictionary` or `Mapping` in general with each argument (member of the original collection) mapped to the result of applying the block to it.

3.8.1 Extensible Collections

Collections derived from `ExtensibleCollection` can be modified by adding or removing elements in various ways. The core protocol is:

add: inserts the given object into the collection.

remove: removes an object equal to the given one from the collection.

addAll: inserts all elements from the first collection contained in the second.

removeAll: removes all elements from the first collection contained in the second.

clear removes all the elements from the collection.

3.8.2 Sequences

`Sequences` are `Mappings` from a range of natural numbers to some objects, sometimes restricted to a given type. Slate sequences are all addressed from a base of 0.

at: answers the element at the index given.

at:put: replaces the element at the index given with the object that is the last argument.

doWithIndex: operates as do:, but applying the block with both the element as first argument and the index of the element in the Sequence as the second argument.

Arrays `Arrays` are fixed-length sequences of any kind of object and are supported primitively. Various parameter types of `Array` are supported primitively, such as `WordArray`, `ByteArray`, and `String`.

Vectors `Vectors` and `Tuples` are fixed-length sequences constructed for geometrical purposes. `Points` happen to be `Tuples`. The constructor message for these types is "`,`".

Subsequences / Slices `Subsequences` allow one to treat a segment of a sequence as a separate sequence with its own addressing scheme; however, modifying the subsequence will cause the original to be modified.

Cords `Cords` are a non-copying representation of a concatenation of `Sequence`s. Normal concatenation of Sequences is performed with the `;` method, and results in copying both of the arguments into a new `Sequence` of the appropriate type; the `;;` method will construct a `Cord` instead. They efficiently implement accessing via `at:` and iteration via `do:`, and `Cord as: Sequence` will "flatten" the `Cord` into a `Sequence`.

Extensible and Sorted Sequences An `ExtensibleSequence` is an extensible `Sequence` with some special methods to treat both ends as queues. It provides the following additional protocol:

`addFirst:` inserts the given object at the beginning of the sequence.

`addLast:` inserts the given object at the end of the sequence.

`add:` inserts the given object at the end of the sequence (it's addLast:).

`first:` answers a sequence of the first N elements.

`last:` answers a sequence of the final N elements.

`removeFirst` removes the first element from the sequence.

`removeLast` removes the final element from the sequence.

A `SortedSequence` behaves similarly except that it will arrange for its members to remain sorted according to a block closure that compares two individual elements; as a result, it should not be manipulated except via `add:` and `remove:` since it maintains its own ordering. A `Heap` is a `SortedSequence` designed for collecting elements in arbitrary order, and removing the first elements.

Stacks A `Stack` is an `ExtensibleSequence` augmented with methods to honor the stack abstraction: `push:`, `pop`, `top`, etc.

Ranges A `Range` is a `Sequence` of `Number`s between two values, that is ordered consecutively and has some stepping value; they include the `start` value and also the `end` value unless the stepping doesn't lead to the `end` value exactly, in which case the last value is the greatest value in the sequence that is still before the end marker value. Creating ranges is identical to the numeric iteration protocol in 3.5.4, without the final `do:` keyword, and sending the `do:` message to a `Range` performs the identical function.

Buffers A `RingBuffer` is a special `ExtensibleSequence` that takes extra care to only use one underlying array object, and also stores its elements in a "wrap-around" fashion, to make for an efficient queue for Streams (see `ReadBufferStream` and `WriteBufferStream` (3.9.2 on page 47)). One consequence of this is that a `RingBuffer` has a limited upper bound in size which the client must handle, although the capacity can be grown explicitly.

3.8.3 Strings and Characters

`Strings` in Slate are non-extensible, mutable `Sequence`s of `Character`s (although `ExtensibleSequence`s can easily be made for them via, say, `as:`). `String`s and `Character`s have a special literal syntax, and methods specific to dealing with text; most of the useful generic methods for strings are lifted to the `Sequence` type. Strings provide the following specific protocol:

`lexicographicallyCompare:` performs an in-order comparison of the codes of the constituent `Character`s of the `String` arguments, returning the sign of comparison, `+1` if the first argument is greater, `0` if equal, and `-1` if lesser.

`capitalize` uppercases in-place the first `Character` in the `String`.

`toUppercase` uppercases in-place all `Character`s in the `String`.

`toLowercase` lowercases in-place all `Character`s in the `String`.

`toSwapCase` toggles in-place the case of all `Character`s in the `String`.

`toCamelCase &separators:` splits the `String` up according to the given separators (defaulting to whitespace), joining `capitalize`'d versions of each element.

`fromCamelCase &separator:` splits the `String` up according to capitalization transition boundaries and joins the elements together with the given separator.

`escaped` answers a new `String` based on adding slash-escapes for literal printing so it can be read in as the same value.

`unescaped` answers a new `String` based on removing slash-escapes, the same way that parsing is done.

readFrom: takes a `String` or a `Stream` with compatible contents and parses the first available data as a `String` with the relevant escape interpretation.

plural uses English rules to answer a new regular plural form of the argument `String`.

asAn uses English rules to answer a new String with `'a '` or `'an '` prepended.

3.8.4 Collections without Duplicates

`NoDuplicatesCollection` forms a special protocol that allows for extension in a well-mannered way. Instead of an `add:` protocol for extension, these collections provide `include:`, which ensures that at least one element of the collection is the target object, but doesn't do anything otherwise. Using `include:` will never add an object if it is already present. These collection types still respond to `add:` and its variants, but they will behave in terms of the `include:` semantics.

The default implementation of this protocol is `Set`, which stores its elements in a (somewhat sparse) hashed array.

3.8.5 Mappings and Dictionaries

`Mapping`s provide a general protocol for associating the elements of a set of keys each to a value object. A `Dictionary` is essentially a `Set` of these `Association`s, but they are generally used with `Symbol`s as keys.

`Mapping` defines the general protocol `at:` and `at:put:` that `Sequence`s use, which also happen to be Mappings. Mappings also support iteration protocols such as `keysDo:`, `valuesDo:`, and `keysAndValuesDo:`.

3.8.6 Linked Collections

A `LinkedCollection` provides a type of collection where the elements themselves are central to defining what is in the collection and what is not.

Linked Lists The usual `LinkedList` type, comprised of individual `Link`s with forward and backward directional access, is provided as a flexible but basic data structure. This type does require that its elements follow the `Link` protocol, however, so it does require some advanced preparation to use it.

Trees Slate includes libraries for binary trees, red-black trees, trees with ordered elements, and tries.

Graphs A directed graph, or `Digraph` (directed graph) type, is provided with associated `Node` and `Edge` types. A `KeyedDigraph` provides the same behavior with a keyed access, similar to that in a `Mapping`, although there is an allowance for various kinds of non-determinism, which makes this useful for creating Nondeterministic Finite Automata.

3.8.7 Vectors and Matrices

Slate includes the beginnings of a mathematical vector and matrix library. See the '`src/lib/matrix.slate`' file for an overview.

3.9 Streams and Iterators

Streams are objects that act as a sequential channel of elements from (or even *to*) some source.

3.9.1 Basic Protocol

Streams respond to a number of common messages. However, many of these only work on some of the stream types, usually according to good sense:

`next &onExhaustion:` reads and answers the next element in the stream. This causes the stream reader to advance one element. If no further elements are available to return, then the block supplied to `&onExhaustion:` is invoked (with no arguments), or if the optional argument was not supplied, an `Exhaustion` condition is signalled.

`peek &onExhaustion:` reads and answers the next element in the stream. This does *not* advance the stream reader. If no further elements are available to return, the `&onExhaustion:` argument is processed as for `next`.

`next:` draws the next `n` number of elements from the stream and delivers them in a `Sequence` of the appropriate type. If fewer than n elements are available before the end of stream (when `next` would have signalled `Exhaustion`), then a shorter `Sequence` than asked for, possibly even empty, is returned.

`next:putInto:` reads the next N elements into the given `Sequence` starting from index 0. Returns the number of elements actually read and inserted. As for `next:`, the returned number may be smaller than actually asked for.

`next:putInto:startingAt:` reads the N elements into the given `Sequence` starting from the given index. Returns the number of elements read and inserted, exactly as for `next:putInto:`.

`nextPutInto:` reads into the given `Sequence` the number of elements which will fit into it. Returns a result exactly as for `next:putInto:`.

nextPut: &onExhaustion: writes the object to the stream. If the stream is full, and will not accept the object, the `&onExhaustion:` optional argument is processed as for `next`.

nextPutAll: *alias* **stream ; sequence** writes all the objects in the `Sequence` to the stream. The **;** selector allows the user to cascade several sequences into the stream as though they were concatenated. If the stream fills up (signalling `Exhaustion`) during the write, a `PartialWrite` condition containing the number of successfully written elements is signalled.

do: applies a block to each element of the stream.

flush synchronizes the total state of the stream with any pending requests made by the user.

isAtEnd answers whether or not the stream has reached some input limit: returns true if next or peek would signal Exhaustion at the time `isAtEnd` was called; returns false otherwise (modulo signalled Conditions). Note that, for some streams (notably sockets), even though `isAtEnd` has returned false, `next` or `peek` may still signal Exhaustion because the underlying input-source has changed its state asynchronously. Use the `&onExhaustion:` optional argument to `next` and `peek` for reliable notifications in these cases.

hasAnEnd answers whether the stream has a definable or possible end-point. All stream types support this protocol and will often recurses on their components to find the answer.

isWritable answers whether the stream is prepared to accept further objects via `nextPut:`. Returns false if `nextPut:` would signal `Exhaustion` at the time `isWritable` was called; returns true otherwise (modulo signalled `Conditions`). See caveat as for `isAtEnd`.

upToEnd collects all the elements of the stream up to its limit into an `Extensible-Sequence`, and returns the sequence.

contents answers a collection of the output of the argument `WriteStream`.

3.9.2 Basic Stream Variants

Figure 3 shows the major stream types and their relationships.

Stream provides the basic protocol for instantiating streams.

ReadStream provides the basic protocol for input access from a source.

WriteStream provides the basic protocol for output access to a target.

ReadWriteStream provides the basic protocol for both read and write access, and caches its input as necessary.



Figure 3: Stream Inheritance

PeekableStream extends `Stream` to provide the ability to look forward on the stream without advancing it.

PositionableStream extends `PeekableStream` to provide a basic protocol to iterate over a sequence of elements from a `Sequence` or a file or other source. These streams store their position in the sequence as they iterate, and are repositionable. It also has its own variants, `-Read-`, `-Write-`, and `-ReadWrite-`.

DummyStream is a (singleton) `ReadWriteStream` that just answers `Nil` repeatedly (and does nothing on writing). It is best created by applying the `reader`, `writer`, or `iterator` methods to `Nil`.

EchoStream is a wrapper for a `Stream` which copies all stream input/output interactions to another `Stream` for logging purposes. It is best created by applying the `echo` (goes to the `Console`) or `echoTo: anotherStream` methods to any stream. It answers the original stream, so that further processing can be chained.

Method ReadStream is a `ReadStream` that targets a no-input block and returns its output each time. It is best created by applying the `reader` or `iterator` method to any block.

Method WriteStream is a `WriteStream` that targets a single-input block and applies its input each time. It is best created by applying the `writer` method to any block.

StreamProcessor is an abstract kind of `ReadStream` that has a source `ReadStream` which it processes in some way. Derivatives specialize it in various useful ways.

FilterStream is a `StreamProcessor` that returns the elements of a wrapped `ReadStream` that satisfy the logical test of a single-argument block being applied to each element. It is created by applying the `select:` or `reject:` method to any `Stream`.

CollectStream is a `StreamProcessor` that returns the results of applying a single-argument block to each element of a `ReadStream` that it wraps. It is created by applying the `collect:` method to any `Stream`.

InjectStream is a `StreamProcessor` that returns the results of successively applying a two-argument block to an initial seed and each stream element. The result of the block is used as seed for the next block application. It is created by applying the `inject:into:` method to any `Stream`.

Tokenizer is a `StreamProcessor` that returns the sequences of elements between a given set of separators. It is created by applying the `split` (using whitespace separators) or `splitWith:` methods to any `Stream`.

GeneratorStream is a `ReadStream` that returns the results of successively applying a single-argument block to an initial seed. The result of the block is used as seed for the next block. The method `generate:until:` creates such a stream with a termination block which takes each element as the argument to determine where the stream should end.

ReadBufferStream wraps another stream with a special `Buffer` object for reading large chunks from the stream at a time while handing out the elements as requested. This also minimizes stress on the memory-allocator by avoiding unnecessary allocation of arrays. It is created by applying the `readBuffered` method to any `Stream`.

WriteBufferStream wraps another stream with a special `Buffer` object for writing large chunks to the stream at a time while accepting new elements as requested. This also minimizes stress on the memory-allocator by avoiding unnecessary allocation of arrays. It is created by applying the `writeBuffered` method to any `Stream`.

3.9.3 Basic Instantiation

There are a number of ways to create `Stream`s, and a large number of implementations, so some methods exist to simplify the process of making a new one:

newOn: creates a new `Stream` of the same type as the first argument, targeting it to the second as a source. This should not be overridden. Instead, the re-targeting method `on:` is overridden.

newTo: creates a new `WriteStream` of the appropriate type on the specified target. This should be overridden for derived types, and the first argument should apply to the generic `Stream` type to allow any instance to know this protocol.

newFrom: creates a new `ReadStream` of the appropriate type on the specified target. This should be overridden for derived types, and the first argument should apply to the generic `Stream` type to allow any instance to know this protocol.

buffered creates and returns a new `BufferStream` whose type corresponds to the argument and wraps the argument `Stream`.

readBuffered creates and returns a new `ReadBufferStream` which wraps the argument `Stream`.

writeBuffered creates and returns a new `WriteBufferStream` which wraps the argument `Stream`.

echoTo: creates and returns a new `EchoStream` which wraps the first argument `Stream` and echoes to the second.

echo creates and returns a new `EchoStream` to the `Console`.

>> performs a looping iterative transfer of all elements of the first stream to the second. The second argument may be any `WriteStream`, or a `StreamProcessor`, or a single-argument `Method` in which case it has the same semantics as `collect:`. For targets to `ExternalResource`s, it will perform a buffered transfer. This method always returns the target stream so that the results may be further processed.

3.9.4 Collecting Protocols

Mirroring the collection protocols, streams support a mirror of that interface (`do:`, `select:`, `collect:`, `reject:`, `inject:into:`). The difference is that where collections would answer other collections, streams return corresponding streams.

3.9.5 Iterator Streams

Many types (typically collections) define their own `Stream` type which goes over its elements in series, even if the collection is not ordered, and only visits each element once. This type's prototype is accessed via the slot `ReadStream` within each collection (located on its traits object). So "`Set ReadStream`" refers to the prototype suitable for iterating over `Set`s.

In order to create a new iterator for a specific collection, the `iterator` message is provided, which clones the prototype for that collection's type and targets it to the receiver of the message. The protocol summary:

iterator will return a `ReadStream` or preferably a `ReadWriteStream` if one is available for the type, targetted to the argument of the message.

reader and **writer** get streams with only `ReadStream` and `WriteStream` capabilities for the type, when available, targetted to the argument of the message.

The stream capabilities supported for each basic collection type are usually limited by the behavior that the type supports. The capabilities per basic type are as follows; types not mentioned inherit or specialize the capabilities of their ancestors:

Type	Capabilities
Collection	none
ExtensibleCollection	Write
Bag	Read and Write separately
Sequence	Positionables (R, W, RW); copy for extension
ExtensibleSequence	Positionables (R, W, RW)

3.10 External Resources

Slate includes an extensible framework for streams that deal with external resources, such as files or network connections or other programs. This generally relies on having a representative object in the image which tracks the underlying primitive identity of the resource, and also provides methods for iterator-style streams over what is available through the resources. Many of these resources aren't sequences as files are sequences of bytes, so they're asynchronous and behave differently from ordinary streams.

Basic Types

`ExternalResource` provides the basic behavior for external resource types.

`ExternalResource Locator` provides a core attribute type for structured descriptors of external resources, as a generalization of file pathnames, port descriptions, URLs, or even URIs.

`ExternalResource Stream` provides an extension of the iterator interface (described on 3.9.5 on the preceding page) for `ExternalResource`s. All of them provide `ReadStream`, `WriteStream`, and `ReadWriteStream` variants as appropriate.

Primitives Extending the framework to cover new primitive or otherwise connection types is fairly simple, since the following methods are the only input/output primitives needed for defining an external resource type:

`resource read: n from: handle startingAt: start into: array` reads the next n elements from the resource identified by the given low-level handle, from the given starting point. The contents are placed in the given array, which should be a `ByteArray` currently.

`resource write: n to: handle startingAt: start from: array` writes the next n elements to the resource identified by the given low-level handle, from the given starting point. The contents are read from the given array, which should be a `ByteArray` currently.

Standard behavior

open Opens the resource for usage within the system.

close Closes the resource, releasing related administrative data; this happens automatically during garbage collection, but it is poor practice to rely upon this.

enable Creates the external resource represented (used by `open`).

isOpen answers whether the resource is open or closed.

isActive answers whether the resource is active.

restart restarts the resource if it's already active.

flush flushes any unwritten elements.

commit commits all pending write-out information to the resource. Commit performs a flush but also ensures that the data is actually sent to the peer.

read:startingAt:into: sends `read:from:startingAt:into:` with the resource's handle.

write:startingAt:from: sends `write:to:startingAt:from:` with the resource's handle.

interactor returns a `ReadWriteStream` for accessing the resource. Unlike the stream that `iterator` returns, `interactor` is expected to return a coupled pair of a `ReadStream` and `WriteStream` over the same resource, synchronized to preserve the resource's behavior.

bufferSize answers a sensible buffer size for interaction, possibly dynamically determined.

defaultBufferSize answers a default sensible buffer size for interaction.

locator answers a suitable structured object for dealing with that resource's identity/location.

sessionDo: executes a code block with the resource as its argument, opening and closing the resource transparently to the block, even for abnormal terminations. If the resource is already open, then this takes no opening or closing actions, so that calls may be nested blindly.

3.10.1 Consoles

The Slate interpreter provides two console `Streams` primitively, `ConsoleInput` and `ConsoleOutput`, which are `Read-` and `WriteStreams` by default, capturing keyboard input and writing out to the console, respectively. These are also accessible as `Console reader` and `Console writer`. `Console interactor` delegates to these, acting as a `ReadWriteStream`.

3.10.2 Files

Files are persistent external sequences of bytes. The interpreter provides an object type `File` which provides the corresponding protocol extensions to `ExternalResource`:

`newNamed:&mode:` returns a new `File` with the given name as its locator and also a mode option. No attempt to open the file is automatically made.

`sessionDo:&mode:` extends `sessionDo:` with a simple temporary mode option (which is reset on return).

`withOpenNamed:do:&mode:` wraps `sessionDo:` with the ability to create a new `File` dynamically for the session along with a specified mode.

`position` returns the position within the `File` in byte units.

`position:` sets the position within the `File` to the given integer number of bytes.

`size` returns the `File` size in bytes.

`name` returns the `File`'s pathname.

`fullName` will always return a complete pathname whereas the regular method may not.

`renameTo:` adjusts the file to have the given name.

`isAtEnd` answers whether the file's end has been reached.

`create` makes a file with the given name, with empty contents.

`exists` answers whether there is a file with the object's pathname.

`delete` deletes the file.

Perhaps the most important utility is to load libraries based on path names. `load: 'filename'` will execute a file with the given path name as Slate source.

File mode objects specify the interaction capabilities requested of the underlying system for the handle. The modes consist of `File Read`, `File Write`, `File ReadWrite`, and `File CreateWrite`.

3.10.3 Shells and Pipes (Not Currently Implemented)

The `Shell` and `Environment` globals are provided to access the underlying operating system's command shell functionality from within the Slate environment. `Shell` provides a dispatch hook for shell-related methods, while `Environment` acts as a `Dictionary` of the current shell context's defined variables and values. They support several primitive methods as follows:

Shell enter enters the host operating system's command-line shell. This has no effect on the Slate environment directly except for suspending and resuming it.

Shell runProgram: programName withArgs: argsArray executes the program with the given name and passes it the arguments which must be `Strings`.

Shell execute: scriptFilename passes the contents of the file named along to the shell, returning its output.

Environment at: varName returns the value stored in a given environment variable.

Environment at: varName put: newValue stores the new value into the given environment variable named.

Environment keys returns an `Array` of the environment variable names defined in the context that Slate was executed in. This result is independent of the actual environment variables.

Environment values returns an `Array` of the environment variable values in the context that Slate was executed in, in the same order as the keys are returned. This result is independent of the actual environment variables.

3.10.4 Networking

`Socket`s are `ExternalResources` that provide a connection on an operating system port. The behaviors specific to these two types are as follows:

newOnPort: creates a new socket of the appropriate type to listen/request on the port number.

listen &buffer: sets the port to listening for client connections, with an optional override of the system default number of request queue entries.

accept creates a new socket connection for the next incoming request.

shutdown shuts down the socket, called when closing (automatically).

host answers the hostname of the socket.

port answers the port number of the socket.

peerHost answers the hostname of the peer.

peerPort answers the peer's port number.

peerIP answers the Internet Protocol address string of the peer, if any.

status answers a `Symbol` representing the socket's current status.

3.11 Paths

Since the environment is structured into namespaces and many types have so-called "attribute types" (for example, the iterator streams), it is helpful to have more than a name by which to refer to an object or its location in the system. The path library allows doing this with `Path` objects, which describe the sequence of slot traversals necessary to reach a particular object. A sub-variant called `RootedPath` remembers the specific origin object used for the path and follows the same protocol.

`Path from: root to: target` creates a new `Path` object containing the necessary slot traversals to obtain the target object (the destination) from the given root (the origin). Paths are created by searching breadth-first through slot names; if the search completes without an identical match, `Nil` is returned.

`to:` works like `from:to:` except that `here` is taken as the origin.

`target` answers the target object as computed by following the slot names from the origin.

`targetFrom:` answers the target object relative to the given one, taking it as the origin.

`reduced` answers another equivalent `Path` object consisting of all non-delegation slots traversed. This gives a "canonical" or shortest path to the object.

`expanded` answers another equivalent `Path` object consisting of all delegation slots required to traverse to reach the object directly.

`root knows: target` answers whether there is a `Path` from the given root to the target.

`isWellKnown` answers whether there is a `Path` from the `lobby` to the given object.

`Path` objects also respond to relevant `Sequence` protocols such as `;` and `isPrefixOf:` among other paths.

3.12 Exceptional Situations and Errors

Slate has a special kind of object representing when an exceptional situation has been reached in a program, called a `Condition`. Condition objects may have attributes and methods like other ordinary objects, but have special methods for dealing with exceptional situations and recovering from them in various ways, often automatically.

3.12.1 Types

`Condition` An object representing a situation which must be handled. This also provides a hook for working with the control-flow of the situation, and dynamic unwinding of control.

`Restart` An object representing and controlling how a condition is handled. Because they are a kind of `Condition`, they can themselves be handled dynamically.

`Warning` A `Condition` which should generate notifications, but does not need to be raised for handling, i.e. no action needs to be taken. Raised by `warn:` with a description.

`StyleWarning` A `Warning` that certain conventions set up by the library author have not been followed, which could lead to problems. Raised by `note:` with a description.

`BreakPoint` A `Condition` that pauses the current computation. Raised by `break` in a context.

`Abort` A `Restart` which unwinds the stack and cleans up contexts after a condition is raised. This is raised by the context method `abort`.

`SeriousCondition` A `Condition` that requires handling, but is not a semantic error of the program. Rather, it's due to some incidental or pragmatic consideration.

`Error` A `SeriousCondition` which involves some misstep in program logic, and raises the need for handlers to avoid a program crash. Raised by `error:` with a description.

3.12.2 Protocol

`signal` Raises the exception that is the argument. This will immediately query for exception handlers in the current context, performing dynamic automatic recovery if possible, or starting the debugger if not.

`on:do:` Executes the block of code with a dynamically bound handler block for the given type of condition.

`ensure:` This is a block method that ensures that the second block is executed either after the first or in *any* case if the original is aborted or control is otherwise handed elsewhere in the middle of execution without possibility of returning into that same execution.

`handlingCases:` Executes the block of code with a set of dynamically bound handler blocks, give as an `Array` of `Association`s between `Condition` objects and the handlers.

return/return: Returns from the condition with a value (default `Nil`), to the point where the condition was signalled.

exit/exit: Aborts from the condition, or aborts from it with a value, to the point where the handler was set up.

defaultHandler This is the condition method that is called if no other handlers are found for the context.

3.13 Concurrency

3.13.1 Processes

Concurrency mostly involves the unit of the `Process`, which is conceptually a manager for a single interpreter. Concurrency may be performed within the context of a single process, using stack / control-flow interaction, or across processes using messaging.

`Process`es in Slate run in turns, each acting as an interpreter with an event loop. They also act as object pools, being opaque and having a context with various local elements which must be explicitly published to be visible to objects in other Processes.

3.13.2 Promises and Eventual-Sends

Coupled with event-loop concurrency is Slate's support of deferred execution, using the `%` prefix for selectors (for the message-send) or for entire expressions. When a deferral is performed on a message-send, we call it an eventual message-send. If the return value is in a syntactic position where it might be used, the caller may expect an object of type `Reference` or `Promise` to be delivered as the answer. These objects are resolvable placeholders, which may also communicate errors in the requested computation.

What's significant is that the use of deferred evaluation means that the relevant expression costs the caller nothing in the immediate execution turn; no matter how expensive side-effecting an expression may be, it will not be executed until at least the next turn in the owning process. Specifically, when a deferred expression is evaluated, it is turned into an event and added to the end of the owner `Process`' queue. When a process finishes each turn, it executes one event at a time from the other end of the queue. Also, if one deferred expression involves the use of the return value from another deferred expression, the dependency will be enforced so that such events will execute in the data-flow partial order in which they were created.

3.14 Types

In coordination with the reserved syntax for type-annotation in block headers, the standard libraries include a collection of representations of primitive TYPES as

well as quantifications over those types. The library of types is laid out within the non-delegated namespace `Types` in the lobby.

3.14.1 Types

`Any` The type that any object satisfies: the universal type.

`None` The type that no object satisfies: the empty type.

`Range` A parametrized type over another type with a linear ordering, such as `Integer`. This type is bounded, it has a `start` and a `finish` (least and greatest possible member). In general, any `Magnitude` can be used as a base of a Range type.

`Member` The type associated with membership in a specific set of objects.

`Singleton` The type of a single object, as distinct from any other object.

`Clone` The type of an object and its CLONE FAMILY, the set of objects that are direct copies (clones plus changes to slot values) of it.

`Array` The type representing all arrays, as parametrized by an element type and a length.

`Block` The type representing code closures of a given (optional) input and output signature.

3.14.2 Operations

Types may be combined in various ways, including `union:`, `intersection:`, and extended via `derive` which preserve type constraints on the derivations.

3.14.3 Type Annotations

Local slot specifiers in a Method header as well as input slot specifiers may have types optionally declared within the header. Within a method declaration expression, the input slots may be redundantly specified in the header as well as in the dispatch expression. However, if this is done, the header's specifier needs to be specified as an input slot and if multiple input slot types are specified, they should be specified in order.

The syntax is similar to that for `@`-based dispatch notation: follow the slot name with the bang character "`!`" and then a type expression, which may be a primitive or derived type. For example,

```
[| :foo!Integer bar | bar: (foo raisedTo: 3).
foo + bar] applyWith: 4.3.
```

Type annotations don't use primitive expressions: the evaluator doesn't have a library of pre-built types at its disposal. Instead, Type annotation expressions are evaluated within the namespace named `Types` accessible from the `lobby`. For this reason, user-defined named types should be installed in some place accessible through the `Types` path.

3.14.4 Type Inference

Type-inference on syntax trees is driven by calling `inferTypes` on the `Syntax Node` in question. This will process type information already annotated to produce derived annotations on related nodes.

Also, there is a facility to extend the type-inference capability per method. To explain, each Type object comes with a `rules` object slot that is dual to the traits delegate object; rules delegate as the traits do but do not confer to the types their methods. Instead, they are used by the inference system transparently to allow for more intelligent specialization. To wit:

```
_@((Member of: {True. False}) rules) ifTrue: then@(Block rules)
   ifFalse: else@(Block rules)
[
  then returnType union: else returnType
].
```

is a type-inference extension method for `#ifTrue:ifFalse:` for any boolean and a pair of blocks, that the return type will be in the union of the blocks' return types.

3.15 Modules

A simple module system is provided, designed to capture the bare essentials of a collection of coherent code. The current module system is just associated with each library file for simplicity's sake. The methods `provides:` and `requires:` applied to the context will add to and check against a global `features` sequence respectively, and missing requirements are noted as the code is loaded. Again for simplicity, `features` currently contains and expects `Symbols`. The `load:` method also invokes a hook to set the `currentModule` in its context.

3.15.1 Types

`Module` a group of objects and methods, along with some information about their definitions. Modules can also provide privacy boundaries, restricting certain methods' accessibility outside of the module.

`FileModule` a module that has been built from source code from a file.

System a collection of modules that together provide some larger service. Systems notably support operations on them to control large-scale libraries.

3.15.2 Operations

Module new creates a new `Module` with no contents.

Module newLocated: creates a new `Module` with the given locator (a filename produces a `FileModule`).

Module newForFileNamed: creates a new `FileModule` for the given file name.

load loads the module or system.

build (re-)builds the module or system.

provide: adds the element to the module's provision collection.

provides: declares a collection's elements to be provided by the current module.

requires: declares a collection's elements to be required by the current context. If any are not found, an error is raised.[8]

import:from: adds an element to the import collection of the current module from the other one's provisions. If it's not provided by the other module, an error is raised.

importAll:from: adds a collection's elements to the import collection of the current module from the other one's provisions. If it's not provided by the other module, an error is raised.

3.15.3 Auto-loading

The `AutoLoader` object manages unloaded `Module`s and allows them to be transparently loaded when their features are requested (via simple message-send). This mechanism relies primarily on defining methods which mimick accessors for the actual resolved object exactly, except for the fact that their action involves loading the file and then re-sending the message once that completes successfully. It is essential that the match between stub and feature be exact, since the feature itself must replace the stub once done. However, the standard protocol hides these mechanics and verification successfully, as follows:

AutoLoader readFromSourceIn: takes a `File Locator`, reads the source in the file, and fills the database with mappings from the features it defines to the module representing it.

AutoLoader performScan &files: takes a list of `File Locator`s, runnung `readFromSourceIn:` on each. By default it scans a default list.

[8] In the future, automatic querying and loading an appropriate module could be added.

AutoLoader installReadyItems reads definitions in the database, installing stubs for those which only take one message to resolve.

AutoLoader writeToStorage writes its definitions to a default external database file for quick re-constitution.

AutoLoader readStorage reads definitions from the default external database file. It will automatically calling `installReadyItems`.

3.16 Persistence

3.16.1 Slate Heap Images

Slate's environment is available for saving as a whole via the representative object `Image`. It offers the following methods:

save &name: Writes out a file containing the contents of the heap, after running a simple cleaning sweep of memory contents. The filename defaults to `'slate.image'` or to the name of the most recent filename parameter value given. Note that execution context state is not disturbed, so saves may occur in any context as a snapshot of system-wide state, including debuggers, inspectors, and other live running processes.

handleShutdown Performs each action (a block) in the `Image`'s `shutdownActions` attribute, a `Dictionary` mapping keys to those corresponding actions. The keys are not meaningful except for identification by the library which installs it, so generally `Symbol`s are used or objects which can be identified and distinguished very easily. `handleShutdown` is actually called by the `quit` method. The purpose of shutdown actions is to clean up any state that may affect the surrounding environment if not handled properly.

handleStartup Performs each action (a block) in the `Image`'s `startupActions` attribute, a `Dictionary` mapping keys to those corresponding actions. The keys are not meaningful except for identification by the library which installs it, so generally `Symbol`s are used or objects which can be identified and distinguished very easily. `handleStartup` is actually called by the `save` method when the image is re-started, because the execution context is preserved, and the VM sends a signal to indicate a fresh startup. The purpose of startup actions is to clean up any state that may be left over from a previous session with that Slate environment, and also to re-initialize state which must persist to avoid inconsistent execution of various services. `ExternalResource`s (3.10 on page 49) whose connections must be persistent are primary users of these hooks.

handleSave Performs each action (a block) in the `Image`'s `saveActions` attribute, a `Dictionary` mapping keys to those corresponding actions. The keys are not meaningful except for identification by the library which installs it, so

generally `Symbol`s are used or objects which can be identified and distinguished very easily. `handleSave` is actually called by the `quit` method. The purpose of save actions is to clean up any state that should not be saved at all; for example, passwords stored for secure access to external resources should be purged. In general, startup actions should be used for any other save-related activities.

3.16.2 Heap Image Segments

We are developing methods for extracting "slices" of images and saving them on disk along with explicit linking information so that groups of arbitrary objects may be transported reliably.

4 Style Guide

Slate provides an unusual opportunity to organize programs and environments in unique ways, primarily through the unique object-centered combination of prototypes and multiple-argument dispatch. This section provides a guide to the generally recommended style of developing in this environment, to promote a better understanding of the system and its usage.

4.1 Environment organization

4.1.1 Namespaces

New namespaces should be used for separate categories of concepts. Occasionally, these are the kind that should automatically included in their enclosing namespace (which can be further inherited up to the lobby). This is done simply by placing the new namespace object in a delegate slot.

4.1.2 Exemplars or Value Objects

These represent objects such as specific colors with well-known names, or cloneable objects with useful default values. Generally these should have capitalized names if they are cloneable, and can be capitalized or lowercase if not. For cases with a large group of such value objects, like colors, there usually should involve a separate namespace to avoid cluttering up the surrounding one. This also helps with naming the use of a value if the intuitive interpretation of its name is dependent on context.

4.2 Naming Methods

One of the primary benefits and peculiarities of the Smalltalk family's style of method syntax is that it provides an opportunity to name one's protocols using

something resembling a phrase. Usually, it is recommended to re-use protocols whenever describing similar behaviors, as an aid to the user's memory in matching functionality to a name to call; in some exceptional situations, different protocols are helpful when there is more than one desired algorithm or behavior to provide for a kind of object. Here are some general practices which have been brought forward from years of Smalltalk practice.

4.2.1 Attributes

Attributes are perhaps the simplest to name of all, in that they are generally nouns or noun phrases of some sort, whether used as direct slots or methods which calculate a property dynamically.

4.2.2 Queries

Methods which test for some fact or property about a single object are generally given a "whether"-style phrase. For example, `myCar isRed` answers whether one's car is red. Slate offers an additional idiom over this particular style, in that `myCar color is: Red` is also possible, since `is:` looks at both the subject and the object of the query. The only remaining obstacle is whether the conceived subject can stand on its own as a named concept; if there are multiple perspectives in normal use, the language can support a certain amount of ambiguity using subjective overrides, but there are limits to this.

4.2.3 Creating

While the method `clone` is the *core* of building new objects in Slate, rather than instantiating a class, there is still the need to provide and use an idiom for delivering optional attributes and varying semantics of creation to one's new objects. Generally, these methods should start with `new-` as a prefix to help the reader and code user to know that the original object will not be modified, and that the result is a new, separate individual. These methods are usually methods with keywords, with each of the keywords describing each option. If the keyword literally names an attribute, the optional-keyword facility is ideal, but if providing a grammatical phrase using prepositions, it is preferable to create a full keyword method.

4.2.4 Performing Actions

The most interesting protocols are akin to commands, where one addresses the objects in question with a phrase that suggests performing some action. This should usually have one key verb for each major component of the action (there is usually just one action per method, but `select:thenCollect:`, for example, performs two), and prepositions or conjunctions to relate the verbs and nouns.

There is a particular idiom of languages without multiple dispatch to add the noun (or type) name of an argument to the name of a function involving it. Slate makes

this unnecessary and an obstacle to polymorphism, since the type of an argument can be specified in a dispatch as needed. Of course, nouns still sometimes have a useful place in a method name, when the noun is not a formal type, but an informal role or usage name.

4.2.5 Binary Operators

These are perhaps the most controversial of any programming language's protocols. In the Smalltalk family of syntax, there are no precedence orderings between operators of different names, so the issues with those do not arise. However, it is very tempting for the library author to re-use mathematical symbols for her own domain, to allow her users to have a convenient abbreviation for common operations. While this benefits the writer of code which uses her library, there are domains and situations that punish the reader of the code that results.

For example, mathematical addition and multiplication symbols, "+" and "*", are generally associative and commutative. That is, repeated calls to these should be able to re-order their arguments arbitrarily and achieve the same result. For example, $3 + 4 + 5 = 4 + 3 + 5 = 5 + 4 + 3$. However, string concatenation (as an example) is not commutative; we cannot re-order the arguments and expect the same result, i.e. "gold"+"fish"="goldfish" , whereas "fish"+"gold"="fishgold" . Because concatenation is associative, however, we can re-use the punctuation style of the semi-colon ";" and achieve intuitive results. This general style of reasoning should be applied wherever this type of operator name re-use could arise.

4.3 Instance-specific Dispatch

Often there are situations whether the user will want to specialize a method in some argument position for a specific object. There are various reasons to do this, and various factors to consider when deciding to do so.

4.3.1 Motivations

Two common patterns where the developer wants to specialize to a single object usually emerge after extended usage. First, there are domain objects which naturally have special non-sharable behavior. For example, `True` is clearly a particular object that helps define the semantics of the whole system, by representing mechanical truth abstractly. In other situations, the same pattern occurs where one has a *universal* concept, or locally an *absolute* concept within a domain.

Second, there are situations whether the user is demonstratively modifying the behavior of some *thing* in order to achieve some *prototype* that behaves in some situation as they desire. Depending on whether the user decides to share this behavior or not, the instance-specific behavior may or may not migrate to some shared `Traits` object. In either case, this is an encouraged use of objects and methods within Slate.

4.3.2 Limitations

There are factors which weigh *against* the use of dispatch on objects with non-shared behaviors. Generally, these just amount to a few simple reasons. First, the behavior will not be shared, which is obvious, but sometimes not clear to the author. Second, the author may mistake an object for its value or attributes, such as `String`s, which are not unique per their value, and so get unexpected results if they dispatch on a `String` instance. The same is true for all literals of that nature, with the exception of `Symbol`s.

The general rule for defining a `method` on an instance which is a lightweight "value" object, is that the object must be reliably re-identifiable, as `Symbol`s are for the language, or through naming paths from the `lobby` or some other object that the user is given access to, such as a method argument. Otherwise, the user must be careful to hang on to the identity of the given object, which offsets any polymorphism gains and exposes implementation details unnecessarily.

4.4 Organization of Source

The nature (and current limitations) of defining objects, relations, and the methods that operate over them require a certain ordering at this point which is worth mentioning. The central point of constraints is the definition of dispatching methods: these methods must have their dispatch targets available at the time they are evaluated. Since there is no late-binding yet of dispatch expressions, generally the basic construction of one's traits and prototype definitions must all occur before defining methods which dispatch to them. The definition needs merely to introduce the actual object that will be used later; other features of the objects, such as the slots and methods defined upon it, are late-bound and will not hinder a method-definition expression.

In general, however, it is recommended to define methods in a *bottom-up* fashion: that more basic utilities should be introduced before the methods that use them. This allows the user (and the author) of the code to read the program sequentially as a document and have some understanding of a program's components when only the name of the component is seen. Of course, this is not always possible, but it helps often enough.

4.5 Type-Annotating Expressions

4.6 Writing Test Cases

Slate includes a port of Smalltalk's SUnit unit-testing framework (implemented in `src/lib/test.slate`), as well as a collection of unit tests (in `tests/*.slate`). Three namespaces are provided in which users may place their `TestCase`s:

testing UnitTests should contain unit-test cases;

testing RegressionTests is for test cases covering bug fixes, for detecting regressions; and

testing BenchmarkTests is to contain test cases for measuring the speed of the system.

There is a method `testing runAllTests` which runs all `TestCase`s found in a recursive search of the UnitTests and RegressionsTests namespaces.

To write your own test cases, add a prototype derived from TestCase to the namespace you've chosen, within the appropriate container - either UnitTests or RegressionTests. For example, suppose you are writing test cases covering Ranges; you might write

```
UnitTests addPrototype: #Range derivedFrom: {TestCase}.
```

Once your prototype has been constructed, add tests to it. Test methods are unary methods with selectors beginning with `test`. For instance,

```
tc@(UnitTests Range traits) testInclusion1
"Verify that all elements of an Range are found in that Range."
[| range |
  range: (25 to: 50 by: 1).
  tc assert: (range allSatisfy: [| :item | range includes: item]).
].
```

The important assertion methods are

```
assert: Boolean
deny: Boolean
should: Method
should: Method raise: Condition
shouldnt: Method
shouldnt: Method raise: Condition
```

and variants with an additional `description: String` argument.

Once all your test methods are defined, a `suite` method should be defined that constructs a `TestSuite` for exercising the `TestCase`:

```
t@(UnitTests Range traits) suite
[t suiteForSelectors: {
  #testInclusion1.
  "... etc. ..."
}].
```

See also the method `TestCase suiteForSelectors:`. The `suite` method is called by `runSuite`, which in turn is called by `runAllTests`.

At this point, invoking `testing runAllTests` will exercise your new code.

References

[Chambers 97] *The Cecil Language: Specification & Rationale*. Craig Chambers. Cecil/Vortex Project. 1997.
Available Online `http://www.cs.washington.edu/research/projects/cecil/www/pubs/cecil-spec.html`

[Salzman 04] *Multiple Dispatch with Prototypes*. Lee Salzman, 2004.
Available Online `http://tunes.org/`

[Ungar et al 95] *The Self Programmer's Reference Manual*. Ole Agesen, Lars Bak, Craig Chambers, Bay-Wei Chang, Urs Holzle, John Maloney, Randall B. Smith, David Ungar, and Mario Wolczko. Sun Microsystems and Stanford University, 4.0 edition, 1995.
Available Online `http://research.sun.com/self/language.html`

[Graham 94] *On Lisp: Advance Techniques for Common Lisp*. Paul Graham. Prentice-Hall, Inc., 1994.
Available Online `http://paulgraham.com/onlisp.html`

[The Refactory] The Smalltalk Refactoring Browser. Online Overview `http://www.refactory.com/RefactoringBrowser/Rewrite.html`

[Smith 96] *The Subjective Prototype*. Randy Smith. Sun Microsystems Laboratories. 1996.
Available Online `http://www-staff.mcs.uts.edu.au/~cotar/proto/randy.txt`

Index

#[] (pound-square brackets), 26
’ (single quote), 24
() (parentheses), 12, 25
& (ampersand), 16, 26
 (asterisk), 26
{} (curly braces), 25
$ (dollar sign), 23
# (pound sign), 24, 25
_ (underscore), 14
+ (plus), 36
, (comma), 41
- (minus), 36
. (period), 12, 14
: (colon), 26
; (message), 53
; (semicolon), 41, 45
;; (message), 41
<- (message), 35
<< (message), 37
= (equal sign), 28
== (message), 28
 == (message), 28
>> (message), 37, 48
(at), 13
,
 (exclamation mark), 19, 56
‘ (back-tick), 19
/(forward slash), 36
/␣(message), 32
‘>> (macro), 21
\char ‘ (message), 32
‘commutatively (macro), 35
‘er (macro), 35
‘quote (message), 20
‘quote: (message), 21
‘unquote (message), 20
‘unquote: (message), 21
‘with:as: (message), 23
‘withSlotsDo: (macro), 22
* (asterisk), 36
** (message), 35
[] (square brackets), 26
abort (message), 54
Abort (object), 54
above:by:do: (message), 34
above:do: (message), 34
abs (message), 37
accept (message), 52
acceptsAdditionalArguments (message), 30
add: (message), 40, 41
addAccessor:for: (message), 29
addAccessorFor: (message), 29
addAll: (message), 40
addDelegate: (message), 8
addDelegate:valued: (message), 8
addFirst: (message), 41
addImmutableSlot:valued: (message), 8
addition, 36
addLast: (message), 41
addMutator:for: (message), 29
addMutatorFor: (message), 29
addPrototype:derivedFrom:, 31
addSlot: (message), 8
addSlot:valued: (message), 8
allSelectorsSent (message), 30
ampersand, 16, 26
and, 32
annotations, type, 56
answer (terminology), 5
Any (type), 56
applyTo: (message), 7
applyWith: (message), 7
applyWith:with: (message), 7
applyWith:with:with: (message), 7
arguments, optional keyword, 16, 26
arguments, rest, 26
arithmetic operations, 36
arity (message), 30
Array (object), 38, 40
Array (type), 56
array indices, 25
array syntax, immediate, 25
array syntax, literal, 25
arrays, 25

as: (message), 29, 36
as:newWithAll: (message), 39
asAn (message), 43
asMethod:on: (message), 14
Association (object), 43
asterisk, 26, 36
at: (message), 40, 52
at:put: (message), 40, 52
atEnd (message), 51
atSlotNamed: (message), 29
atSlotNamed:put: (message), 29
AutoLoader (object), 58

back-tick, 19
backslash (character literal), 23
backspace (character literal), 23
Bag (object), 38
bang character, 19
basic operations, 36
basic types, 36
bell (character literal), 23
below:by:do: (message), 34
below:do: (message), 34
between:and: (message), 37
BigInteger (object), 36
binary messages, 10
binary trees, 43
bit operations, 36
bitAnd: (message), 36
bitNot (message), 37
bitOr: (message), 36
bitShift: (message), 37
bitXor: (message), 36
Block (type), 56
block invocation, 7
block, empty, 7
block, termination with a period, 7
blocks, 6, 26
blocks, code, 13
blocks, compile-time, 26
BlockStream (object), 45
Boolean (object), 32
boolean logic, 32
boolean messages, 32
break (message), 54
BreakPoint (object), 54
Buffer (object), 38, 42
buffered (message), 48
bufferSize (message), 50
BufferStream (object), 45
build (message), 58
ByteArray (object), 36, 38, 40

callers (message), 30
capacity (message), 39
capitalize (message), 42
carriage return (character literal), 23
cascade, message, 21
case sensitivity, 9
caseOf: (message), 33
caseOf:otherwise: (message), 33
Cecil, 5
characters, 23
characters, special, 23
clear (message), 40
clone (message), 27, 28
Clone (type), 56
Cloneable (object), 28
cloning, 6, 28
close (message), 50
closure, lexical, 8
code block, 6
code blocks, 13
code closures, 6
coercion, 29
collect: (message), 39, 47
Collection (object), 38
collection creation, 39
collection iteration, 39
collection properties, 39
collection testing, 38
collections, 38
collections, extensible, 40
CollectStream (object), 45
colon character, 26
comma, 41
commit (message), 50
compile-time blocks, 26
Complex (object), 36
concurrency, 55
Condition (object), 53, 54
conditional evaluation, 32

Console (object), 45, 50
ConsoleInput (object), 50
ConsoleOutput (object), 50
contents (message), 45
converse (message), 35
conversion, 29
conversion, number, 36
copy (message), 28
copying, 28
Cord (object), 38, 41
create (message), 51
creation, collections, 39
creation, streams, 47
curly braces, 25
currentModule (slot), 57
currying, 35

defaultBufferSize (message), 50
defaultHandler (message), 55
define: (message), 31
delegates (message), 8, 29
delegates: (message), 8
delegatesDo: (message), 29
delegation, 8
delegation testing, 28
delete (message), 51
Derivable (object), 28
derive (message), 28, 31, 56
deriveWith: (message), 56
Dictionary (object), 38, 43
Digraph (object), 44
dimensioned units, 38
dispatch (terminology), 6
dispatch targets, 14
dispatch, instance-specific, 62
dispatch, message, 15
dispatch, multiple, 15
dispatch, subjective, 18
dispatch-overriding, 17
division, 36
division, integer, 37
do (message), 7
do: (message), 39, 45
dollar sign, 23
doWithIndex: (message), 40
downTo:by:do: (message), 34
downTo:do: (message), 34
DummyStream (object), 45

each: (message), 39
early return, 12, 33
echo (message), 46, 48
EchoStream (object), 45
echoTo: (message), 46, 48
Edge (object), 44
empty expression, 13
enable (message), 50
English units, 38
ensure: (message), 54
enter (message), 52
Environment (object), 51
equal sign, 28
equality, value, 28
eqv: (message), 32
Error (object), 54
error handling, 53
error: (message), 54
escape (character literal), 23
escaped (message), 42
evaluation, conditional, 32
evaluation, lazy, 32
exceptions, 53
exclamation mark, 19, 56
execute: (message), 52
exists (message), 51
exit (message), 55
exit: (message), 55
expanded (message), 53
exponentiation, 36
expression sequences, 12
expressions, 9
extensible collections, 40
ExtensibleCollection (object), 38, 40
ExtensibleSequence (object), 38, 41
extensions, 5
external resources, 49
ExternalResource (object), 49
ExternalResource Locator (object), 49
ExternalResource Stream (object), 49

factorial (message), 37
False (object), 32
features (slot), 57

File (object), 51
File CreateWrite (object), 51
File Read (object), 51
File ReadWrite (object), 51
File Write (object), 51
FileModule (object), 57
files, 51
FileStream (object), 45
FilterStream (object), 45
findOn: (message)], 17
findOn:after: (message), 17
finish (message), 56
first: (message), 41
Float (object), 36
flush (message), 45, 50
form feed (character literal), 23
forward slash, 36
Fraction (object), 36, 37
fractionPart (message), 37
from:to: (message), 53
fromCamelCase (message), 42
fullName (message), 51

gcd: (message), 37
generate:until: (message), 47
global objects, 26
Graph (object), 38
graphs, 44

handleSave (message), 59
handleShutdown (message), 59
handleStartup (message), 59
handlingCases: (message), 54
hasAnEnd (message), 45
hash (message), 28
hashing, 28
Heap (object), 41
here (message), 27
host (message), 52

identity (terminology), 28
identityHash (message), 28
ifFalse: (message), 33
ifNil: (message), 33
ifNil:ifNotNil: (message), 33
ifNil:ifNotNilDo: (message), 33
ifNotNil: (message)', 33
ifNotNilDo: (message), 33
ifTrue: (message), 33
ifTrue:ifFalse: (message), 33
Image (object), 59
immediate array syntax, 25
implementations (message), 30
implicit-context sends, 11
import:from: (message), 58
importAll:from: (message), 58
include: (message), 43
includes: (message), 38
indices, array, 25
inference, type, 57
inferTypes (message), 57
inheritance (terminology), 6
inject:into: (message), 40, 47
input slots, 7
InsertionSequence (object), 38
instance-specific dispatch, 62
Integer (object), 36
integer division, 37
interactor (message), 50
intern (message), 35
intern: (message), 35
intersection: (message), 56
is: (message), 28
isActive (message), 50
isAtEnd (message), 45
isEmpty (message), 38
isNamed (method), 30
isNegative (message), 37
isOpen (message), 50
isPositive (message), 37
isPrefixOf: (message), 53
isReally: message, 28
isSameAs: (message), 28
isWellKnown (message), 53
isWritable (message), 45
isZero (message), 37
iteration, collections, 39
iterator (message), 46, 48
iterator streams, 48
iterators, 44

KeyedDigraph (object), 44
keys (message), 52

keysAndValuedDo: (message), 43
keysDo: (message), 43
keyword arguments, optional, 16, 26
keyword messages, 11
kind testing, 28

labelled quotation, 21
LargeUnbound (object), 38
last: (message), 41
Layer (object), 18
layer (terminology), 19
layering: (message), 19
lazy evaluation, 32
lcm: (message), 37
Lexer (object), 19
lexical closure, 8
lexical scope, 11
lexicographicallyCompare: (message), 42
limits, numeric, 38
LineNumberedStream (object), 45
Link (object), 43
LinkedCollection (object), 38, 43
LinkedList (object), 38, 43
listen (message), 52
literal array syntax, 25
literal arrays, 25
literal syntax, 23
load (message), 58
load: (message), 51, 57
lobby (object), 26
lobby (slot), 26
local slots, 7
locator (message), 50
lookup semantics, 15
loop (message), 34
looping, 34

macro message-sends, 19
macroCallers (message), 30
macroexpand (message), 19
Magnitude (object), 36
Mapping (object), 38, 40, 43
matrices, 44
max: (message), 37
Member (type), 56
message (terminology), 5
message cascading, 21
message dispatch, 15
message-sends, macro, 19
messages, binary, 10
messages, keyword, 11
messages, precedence, 9
messages, resending, 17
messages, types, 9
messages, unary, 9
Method (object), 28
method (terminology), 5
method composition, 35
method definition, 13
Method ReadStream (object), 46
method template, 13
Method WriteStream (object), 46
methods, 13
methods, dynamic generation, 14
methodsAt: (message), 30
methodsCalling: (message), 30
methodsNamed: (message), 30
methodsNamed:at: (message), 30
min: (message), 37
minus, 36
mod: (mesage), 37
mod: (message), 37
Module (object), 57
modules, 57
multiple dispatch, 15
multiplication, 36

name (message), 35, 51
Namespace (object), 27
namespaces, 60
negated (message), 37
NegativeEpsilon (message), 38
NegativeInfinity (object), 38
new (message), 58
newForFileNamed: (message), 58
newFrom: (message), 47
newline (character literal), 23
newLocated: (message), 58
newNamed:&mode: (message), 51
newOn: (message), 47
newOnPort: (message), 52
newSize: (message), 39
newSizeOf: (message), 39

newTo: (message), 47
newWithAll: (message), 39
next (message), 44
next: (message), 44
next:putInto: (message), 44
next:putInto:startingAt: (message), 44
nextPut: (message), 45
nextPutAll: (message), 45
nextPutInto: (message), 44
Nil (object), 13, 28
Node (object), 44
NoDuplicatesCollection (object), 38
NoDuplicationCollection (object), 43
None (type), 56
NoRole (object), 15, 28
not (message), 32
note: (message), 54
null (character literal), 23
Number (object), 36
number conversion, 36
numeric limits, 38

object (terminology), 5
objects, 6
objects, global, 26
Oddball (object), 27
oddballs, 29
on:do: (message), 54
open (message), 50
operations, arithmetic, 36
operations, basic, 36
operations, bit, 36
optional keyword arguments, 16, 26
optional keywords (in a block header), 17
optional keywords (in a message send), 16
optional keywords (in a method definition), 16
or, 32
ordered elements, trees, 43
overriding, dispatch, 17

parentheses, 12, 25
Parser (object), 19
Path (object), 53
pattern-matching, 23
peek (message), 44
PeekableStream (object), 46
peerHost (message), 52
peerIP (message), 52
peerPort (message), 52
period, 14
Pipe (object), 45
plural (message), 43
plus, 36
Point (object), 41
pop (message), 41
port (message), 52
position (message), 51
position: (message), 51
PositionableStream (object), 45
PositiveEpsilon (object), 38
PositiveInfinity (object), 38
pound sign, 24, 25
power, raising to a, 36
precedence, messages, 9
print (message), 28
printing, 28
printOn: (message), 28
printString (message), 28
project: (message), 40
properties, collections, 39
prototypes (object), 27
provide: (message), 58
provides: (message), 57, 58
push: (message), 41

quo: (message), 37
quotation, labelled, 21
quotes, single, 24
quotient, 37
quoting, 20

raisedTo: (message), 36
Range (object), 38, 42
Range (type), 56
read:from:startingAt:into: (message), 49
read:startingAt:into: (mesage), 50
readBuffered (message), 47, 48
ReadBufferStream (object), 42
reader (message), 46, 48
readFrom: (message), 38, 43
ReadPositionableStream (object), 46

ReadStream (object), 45
ReadWritePositionableStream (object), 46
ReadWriteStream (object), 45
reciprocal (message), 37
reciprocal: (message), 37
red-black trees, 43
reduce: (message), 40
reduced (message), 37, 53
reject: (message), 39, 46
rem: (message), 37
remove: (message), 40, 41
removeAll: (message), 40
removeFirst (message), 41
removeLast (message), 41
removeSlot: (message), 8
renamedTo: (message), 51
requires: (message), 57, 58
resend (message), 17
resending messages, 17
resources, external, 49
rest arguments, 26
rest parameter, 7
restart (message), 50
Restart (object), 54
return (message), 55
return, early, 6, 33
return: (message), 55
RingBuffer (object), 42
role, 13
roles, 6
Root (object), 27
RootedPath (object), 53
rules (slot), 57
runProgram:withArgs: (message), 52

save (message), 59
saveActions (object), 59
scope, lexical, 11
seenFrom: (message), 18, 19
select: (message), 39, 46
selector, 9
selector (message), 30
selector (terminology), 5
selector, of keyword message, 11
Self, 5, 13
semicolon, 41
sends, implicit-context, 11
sendTo: (message), 17
sendTo:through: (message), 17
sendWith: (message), 17
sendWith:with: (message), 17
sendWith:with:with: (message), 17
Sequence (object), 38, 40
SeriousCondition (object), 54
sessionDo: (message), 50
sessionDo:&mode: (message), 51
Set (object), 38, 43
Set ReadStream (object), 48
Shell (object), 51
shutdown (message), 52
shutdownActions (object), 59
SI units, 38
sign (message), 37
signal (message), 54
signature (message), 30
single quotes, 24
SingleFloat (object), 36
Singleton (type), 56
size (message), 39, 51
slash, forward, 36
slot mutability, 8
slot properties, 8
slotNames (message), 29
slots, 6
slots, delegation, 8
slotsDo: (message), 29
SmallInteger (object), 36
Smalltalk, 5, 25, 60
Socket (object), 45, 52
SortedSequence (object), 38, 41
SortedSet (object), 38
space (character literal), 23
special characters, 23
splitWith: (message), 47
square brackets, 26
Stack (object), 41
start (message), 56
startupActions (object), 59
status (message), 52
stop mark, 12, 14
Stream (object), 28, 45
stream creation, 47

StreamProcessor (object), 46
streams, 44
streams, iterator, 48
String (object), 38, 40, 42
strings, 24
style guide, 60
StyleWarning (object), 54
Subject (object), 18
subject (terminology), 19
subjective dispatch, 18
Subsequence (object), 41
subtraction, 36
Symbol (object), 35
symbol table, 35
symbols, 24
Syntax Node (object), 19
syntax, literal, 23
System (object), 58

tab (character literal), 23
target (message), 53
targetFrom: (message), 53
testing, collections, 38
testing, delegation, 28
testing, kind, 28
times, 36
timesRepeat: (message), 34
to: (message), 53
toCamelCase (message), 42
toLowerCase (message), 42
top (message), 41
toSwapCase (message), 42
toUppercase (message), 42
traits, 31
traits (message), 31
Tree (object), 38
trees, 43
trees with ordered elements, 43
trees, binary, 43
trees, red-black, 43
tries, 43
True (object), 32
truncated (message), 37
Tuple (object), 38, 41
type annotations, 19, 56
type inference, 57
types, 55
Types (namespace), 55
types, basic, 36

unary messages, 9
underscore, 14
unescaped (message), 42
union: (message), 56
units, dimensioned, 38
units, English, 38
units, SI, 38
unquoting, 20
upTo:by:do: (message), 34
upTo:do: (message), 34
upToEnd (message), 45
Us, 18

value equality, 28
values (message), 52
valuesDo: (message), 43
variable arguments, 7
Vector (object), 41
vectors, 44
vertical feed (character literal), 23

warn: (message), 54
Warning (object), 54
whileFalse (message), 34
whileFalse: (message), 34
whileTrue (message), 34
whileTrue: (message), 34
whitespace sensitivity, 9
withOpenNamed:Do:&mode: (message), 51
withoutLayers (message), 19
withoutSubject (message), 19
WordArray (object), 40
write:startingAt:into: (message), 50
write:to:startingAt:from: (message), 49
writeBuffered (message), 47, 48
WriteBufferStream (object), 42
WritePositionableStream (object), 46
writer (message), 46, 48
WriteStream (object), 45

xor: (message), 32

zero (message), 37