The Slate Programmer's Reference Manual

Brian Rice and Lee Salzman

8th August 2004

Contents

List of Figures

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.

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.

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 some thing in the system that can be identified.

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

Message the act of requesting a behavior or procedure from some objects, the message's arguments.

Answer the response to a message; a value that expressions evaluate into.

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

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

Dispatch the process of determining, from a message-send, what method is appropriate to invoke to implement the behavior.

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 symbols 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 2.3 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 can override this.

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

```
Slate> [| :i j k | j: 4. k: 5. j + k - i].
[]
```

creates and returns a new block. Within the header, identifiers that begin with a colon such as `:i` 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, `j + k - i`. In this block, `i` is an input slot, and `j` and `k` are local slots which are 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 client must know how many and in what order it takes input arguments. Arguments are passed in using one of several messages. By evaluating these messages, the block is immediately evaluated, and the result of the evaluation is the block's execution result.

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. Every block responds properly to `applyTo:` however, which takes an array of the input values as its other argument.

```
Slate> [| :x :y | x quo: y] applyWith: 17 with: 5.
3
Slate> [| :a :b :c | (b raisedTo: 2) - (4 * a * c)]
 applyTo: {3. 4. 5}.
-44
```

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
```

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 Slot Properties

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

Slate provides several primitive messages to manage slots:

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

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

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

`object addDelegate: slotSymbol` and **`object addDelegate: slotSymbol valued: val`** add a delegation slot, and initialize it, respectively. It is recommended to use the latter since delegation to `Nil` is unsafe.

Each of the these has a variant which does not create a mutator method for its slot: `addImmutableSlot:valued:` and `addImmutableDelegate:valued:`.

2.2 Expressions

Expressions in Slate mostly consist of message-sends to argument objects. The left-most argument is not considered an implicit receiver as it is with most message-passing languages, however.

An important issue is that every identifier 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 identifiers in order for them to be considered separate. For example, `ab+4` will be treated as one identifier, but `ab + 4` is a message-send expression.

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 message-sends.
3. Binary message-sends.
4. Keyword message-sends.

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 Message-sends

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 Message-sends

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:

---

[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.

```
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:

```
@ [ ] ( ) { } . : ! | ‘ &
```

2.2.3 Keyword Message-sends

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 Expression Sequences

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. (We do not consider this expression good style, but it illustrates the nature of the language.) 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.

2.2.5 Implicit-context Sends

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.

In some cases, it may be necessary to manipulate the context in particular ways. In that case, it can be directly addressed with a loopback slot named `thisContext`, which refers to the current activation. The essence of this concept is that within a block, `x: 4.` is equivalent to `thisContext x: 4.`[2]

---

[2] The current named method as distinct from the context is available as `currentMethod`, and its name is available as `selector`. However, these are dependent on the current implementation of Slate, and so may not be available in the future.

2.3 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.

2.3.1 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 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 symbol "`_`" 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. 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).

2.3.2 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 interpreter to compile a method using a certain symbolic name and a sequence of objects that are used for dispatching targets. For example:

```
[| :x :y | 5] asMethod: #+ on: {2. 2}.
```

and

```
_@2 + _@2 [5].
```

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}.
```

2.3.3 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 uses the most specific supertype to determine specificity (or rather, the most specific parameter type of which the argument is a subtype), specificity is instead interpreted as DISTANCE in the directed graph of delegations, starting from the argument as the root.

The DISTANCE notion has the following properties:

- It is determined by a depth-first traversal over the delegate slots, considering most-recently-added delegates before previously-added 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.

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), where `NoRole` behaves like an "omega distance", as far away as possible.

2.3.4 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, and when not provided will start with a `Nil` value.

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

An optional keyword argument is passed to a method by forming keyword-value pairs as "`&keywordName: someValue`". 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.

A block can declare optional input keywords in its block header, using “`&argName`” as an input variable declaration, called with the normal convention, whenever the block is invoked with a message-send.

2.3.5 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. The result of the `resend` message is the returned result of that method.

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

`methodName findOn: argumentArray after: aMethod` locates 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.

`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.

`methodName sendTo: argumentArray through: dispatchArray` is an extra option to specify a different signature for the method than that of the actual argument objects.

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

2.3.6 Subjective Dispatch

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. 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.
- 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.

2.4 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 3.11 on page 44.

2.5 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 Nodes`. 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.

2.5.1 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.

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

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.

---

[3] We may also provide these as `'up` and `'down`, respectively, if there is enough demand for it, and it is not too confusing.

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.

2.5.3 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.

2.5.4 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.

2.6 Literal Syntax

2.6.1 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.

2.6.2 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.

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

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"
```

2.6.3 Symbols

Symbols start with the pound sign character (#) and consist of all following characters up to the next non-escaped whitespace, unless the pound sign is followed exactly by a string literal, in which case the string’s contents become the identifier for the symbol. So, for example, the following are all valid symbols and symbol literals:

```
#$
#key:word:expression:
#something_with_underscores
#’A full string with a \nnewline in it.’
#’@’ "@ and other reserved characters must be escaped to deal with their lexical
```

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 symbols, and uses individual context objects to hold local bindings.[4]

2.6.4 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}
```

2.6.5 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,

---

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

```
[| 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.

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.

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

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 `Booleans`, as well as literals (value-objects) such as `Characters` and `Symbols`. 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".

`Derivable` `Derivable` objects respond to `derive` and `deriveWith:`, which means they can be readily extended.

`Cloneable` `Cloneable` objects are derivables that can be cloned.

`Method` A `Cloneable` object with attributes for supporting execution of blocks 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 `print` returns a printed representation of the object. `printOn:` places the result of printing onto a designated `Stream`. This should be overridden for clarity.

Delegation-testing `isReally:` returns whether the first object has the second 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. What's more important about hashing is that 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.

Slot-enumeration For each object, the `Symbols` naming its slots and delegate slots can be accessed and iterated over, using the accessors `slotNames` and `delegateNames`, which work with the symbol names of the slots, or the iterators `slotsDo:` and `delegatesDo:`, which iterate over the stored values themselves.

3.2.2 Oddballs

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

3.3 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.

Traits objects also have their own traits object, which is `Traits traits`. This has the important methods defined on it for deriving new prototypes with new traits objects:

`myObject derive` will return a new clone of the object with a traits object which is cloned from the original's traits object, and an immutable delegation slot set between the traits objects.

`myObject deriveWith: mixinsArray` will perform the same operation, adding more immutable delegation links to 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 `mixinsArray` do not have to be `Derivable`.

`obj addPrototype: name derivedFrom: parentsArray` will perform the effects of either `derive` or `deriveWith:` using all the elements of the `Sequence` in the same order as `deriveWith:`. It also assigns the name to the traits object's name attribute 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.

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

3.4 Closures, Booleans, and Control Structures

3.4.1 Boolean Logic

Slate's interpreter 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

3.4.2 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[5]. Blocks that evaluate logical expressions can be used lazily in these logical expressions. For example,

```
(x < 3) and: [y > 7].
```

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

---

[5] 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) or: [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 make it positive 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.

3.4.3 Looping

Slate includes various idioms for constructing basic loops.

`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.

`whileTrue` and **`whileFalse`** execute their blocks repeatedly, checking the return value before repeating iterations.

`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.

Slate's looping control structures can easily be extended without concern due to the fact that the interpreter unrolls properly tail-recursive blocks into low-level loop code that re-uses the same activation frame. So basically structuring custom looping code so that it calls itself last within its own body and returns that value will avoid the need for increasing stack space per iteration.

3.5 Magnitudes and Numbers

3.5.1 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 `Integers`, however: instead of overflows, `BigInteger` objects of the appropriate value are returned.

`BigInteger` larger `Integers`, implemented as `WordArrays`.

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

`Float` A low-level floating-point numeric representation, being inexact.

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

3.5.2 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.

3.5.3 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 its 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.

3.5.4 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.

3.5.5 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/dimensioned.slate`' file for an overview.

3.6 Collections

Slate's collection hierarchy makes use of multiple delegation 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.

Figure 1 shows the overview of the collection types, and how their delegation 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.

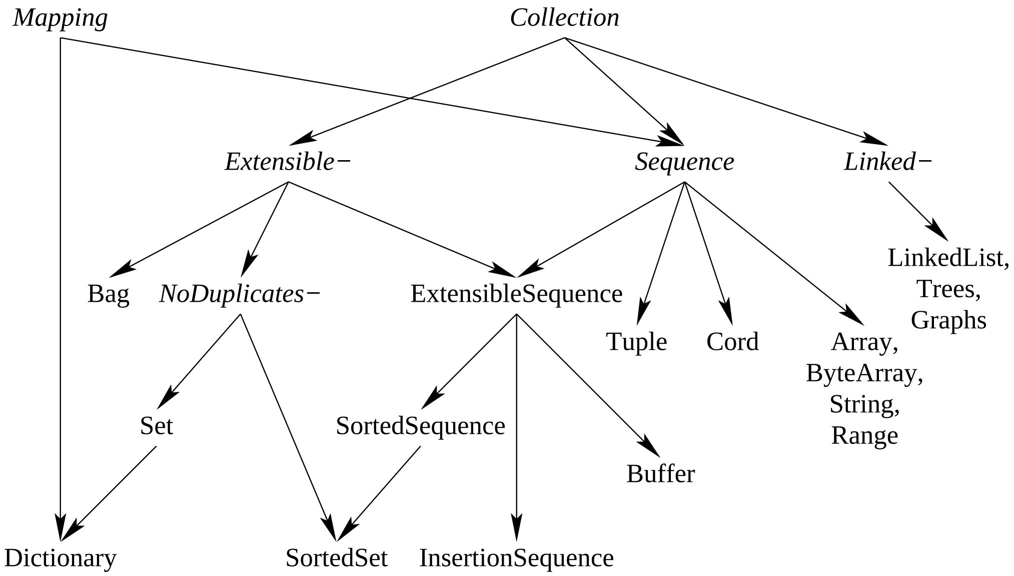

Figure 1: Core Collections Inheritance

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

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

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

`newEmpty` answers a new collection of the same type that is sized to some small default value.

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

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.

3.6.1 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.

3.6.2 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.

To access and modify sequences, the basic methods `seq at: index` and `seq at: index put: object` are provided.

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 `Sequences`. 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 `Numbers` 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.

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 `BufferReadStream` and `BufferWriteStream` (3.7.2 on page 37)). 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.

3.6.3 Strings and Characters

`Strings` in Slate are non-extensible, mutable `Sequences` of `Characters` (although `ExtensibleSequences` can easily be made for them via, say, `as:`). Strings and Characters 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.

3.6.4 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.

3.6.5 Mappings and Dictionaries

`Mappings` 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 `Associations`, but they are generally used with symbols as keys.

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

3.6.6 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 `Links` with forward and backward directional access, is provided as a flexible but basic data structure.

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.

3.6.7 Vectors and Matrices

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

3.7 Streams and Iterators

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

3.7.1 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` reads and answers the next element in the stream. This causes the stream reader to advance one element.

`peek` reads and answers the next element in the stream. This does *not* advance the stream reader.

`next:` draws the next `n` number of elements from the stream and delivers them in a `Sequence` of the appropriate type.

`next:putInto:` reads the next N elements into the given `Sequence` starting from index 0.

`next:putInto:startingAt:` reads the N elements into the given `Sequence` starting from the given index.

`nextPutInto:` reads into the given `Sequence` the number of elements which will fit into it.

`nextPut:` writes the object to the stream.

`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.

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 limit.

upToEnd collects all the elements of the stream up to its limit into an `ExtensibleSequence`.

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

3.7.2 Basic Stream Variants

Figure 2 shows the major stream types and their relationships.

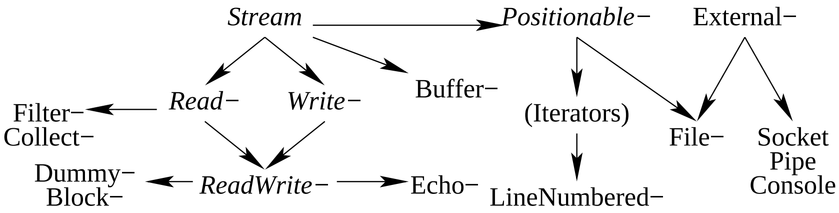

Figure 2: Stream Inheritance

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.

PositionableStream extends `Stream` 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`.

`BufferReadStream` 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`.

`BufferWriteStream` 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`.

3.7.3 Basic Instantiation

There are a number of ways to create `Streams`, 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, targetting it to the second as a source. This should not be overridden. Instead, the re-targetting 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 `ExternalResources`, it will perform a buffered transfer. This method always returns the target stream so that the results may be further processed.

3.7.4 Collecting Protocols

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

3.7.5 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 `Sets`.

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.

`reader` and **`writer`** get streams with only `ReadStream` and `WriteStream` capabilities for the type, when available.

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)

3.8 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.

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

`write:startingAt:from:` calls `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.

3.8.1 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`.

3.8.2 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.

`open:` returns a file handle for a `String` that names a path to a file, for read/write access.

`openForInput:` returns a handle for reading an existing file of the given name, or `Nil` if it doesn't exist.

`openForOutput:` returns a handle for writing (appending) to a file with the given name.

`openNew:` returns a handle for writing (appending) to a new file with the given name. It will create a new file, but if the file exists, `Nil` will be returned.

`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.

`atEnd` 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`.

3.8.3 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.

3.8.4 Networking (Not Currently Implemented)

`SocketClient` and `SocketServer` 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.

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

`wait` waits indefinitely for a connection on the `SocketServer`.

`wait:` waits a specific given amount of time in seconds for a connection on the `SocketServer`. An `Error` is raised if this does not happen. A timeout of zero will cause polling.

`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.

3.9 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.

3.9.1 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.

3.9.2 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.

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

`return/return:` Returns from the condition, or returns from it with a value, 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.

3.10 Concurrency (Not Yet Implemented)

3.10.1 Processes

3.10.2 Scheduling

3.10.3 Synchronization

3.11 Types

In coordination with the reserved syntax for type-annotation in block headers, Slate's 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.

3.11.1 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 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.

3.11.2 Operations

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

3.11.3 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.

3.11.4 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.

3.12 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.

3.12.1 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.

3.12.2 Operations

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

`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.[6]

`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.

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 Slate, 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.

---

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

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 that Slate uses which have been common in Smalltalk practice for years.

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.

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 an idiom for delivering optional attributes 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 keyword methods, with each of the keywords describing each option, whether literally naming an attribute, or simulating a grammatical phrase using prepositions.

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.

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 emerge from using Slate. 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 `Strings`, 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 `Symbols`.

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 `Symbols` 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 what other methods are defined upon it, are late-bound and will not hinder a method-dispatch 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

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/\~{}eihrul/pmd.pdf`

[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`