Chapter 10

Smalltalk and object-orientation

Contents

In 1971, Alan Kay convened a group of researchers at Xerox Palo Alto Research Center to try to realize his idea of a computer that might deserve to be called "personal." Such a computer should be no larger than a notebook, have a friendly interface that could respond to gestures, and offer the reach and expressive power of the best programming languages of the day, like Simula 67. At PARC, Kay's colleagues were inventing and refining the one-person computer, the bitmapped display, the user-interface elements that we now call windows, menus, and pointing, and many other delights that we have long since taken for granted. Kay's group worked to create a software system that would feed Kay's hunger to use the computer as "an amplifier for the human reach," not just a tool used to build software systems (Kay 1993, p70).

For the rest of the decade, Kay's Learning Research Group worked quietly but furiously, creating over 80 versions of their system, which they called Smalltalk. They issued major internal releases in 1972, 1974, 1976, and 1978. In 1980, they made the system public, as Smalltalk-80. In 1981, they published a collection of articles in a special issue of Byte Magazine, which aroused a great deal of interest, but at that time few readers could get access to a Smalltalk system. Today, Smalltalk-80 is freely available in the form of Squeak, a portable implementation that is bootstrapped by translation to C (Ingalls et al. 1997). And object-oriented programming has never been more popular.

Smalltalk's ideas about programming languages and user interfaces influenced countless later systems. We concentrate on the language. Like Clu, Smalltalk makes everything look like a user-defined datatype with associated operations, but Smalltalk provides a very different way of defining such types. A value of user-defined datatype is called an *object.* An object is a thing with which other objects can interact, and many objects represent things real that exist in the real world. Objects interact by sending *messages* to other objects. The object that receives a message is called the message's *receiver.* On receiving a message, an object executes a *method*;[1] the method can do some internal computation and can send messages to other objects, but eventually it replies with an *answer.*

The method that is executed when an object receives a message is determined by the object's *class.* A class is analogous to a Clu cluster, and every object is an *instance* of exactly one class. An object's class determines the set of messages to which that object can respond; this set of messages is the class's *protocol.* Protocols are dynamic; unlike Clu, Smalltalk has no static type system.

A class defines its protocol by associating each message (the interface) with a method (the implementation). A class is something like a Clu cluster without types, but there's a critical difference: a class definition names an existing class as the *parent* of the class being defined. To describe the relationship of a class with its parent, we say that the new class is the *subclass*, and its parent is the *superclass.* The point of having a superclass is that the definition of the subclass need *not* define a method for every message in its protocol: it also has the option to *inherit* methods from its superclass. Once the superclass is named, its methods are inherited automatically, by default. So an instance of the new class automatically knows how to respond to messages defined not only by its own class, but also by its superclass, or indeed by any of its ancestor classes. Like higher-order functions, inheritance provides a powerful way of reusing code. Inheritance is what makes a language "object-oriented," and one of Smalltalk's major contributions was to show the power of inheritance.

[1] For Smalltalk, I say "execute" instead of "evaluate." They mean the same thing, but "execute" carries connotations of imperative features and mutable state, which are typical of most Smalltalk methods.

Smalltalk term	Clu term	Smalltalk meaning
Class	Cluster	Encapsulates the representation of a type of data
Object	Instance	A datum of a particular class
Method	Function	An operation defined in a class
Message	Function name	
Message send	Function application	
Instance variables	Rep	The values representing objects of a class
Protocol	Operations of a cluster	The messages to which an object of a given class can respond

Table 10.1: Correspondence between Smalltalk and Clu terminology

This whirlwind introduction to Smalltalk should help you compare Smalltalk with Clu. For more help, you can find a rough correspondence between Smalltalk terms and corresponding Clu terms shown in Table 10.1. As we make a transition from Smalltalk-80 to μSmalltalk, I highlight two other aspects of Smalltalk and talk about how I model them in μSmalltalk.

- Smalltalk exemplifies the school of language design that takes one idea, simplifies it as much as possible, and pushes it to a logical extreme. Alan Kay (1993) refers to this kind of design as a "crystallization of style." One shorthand for Smalltalk's crystallized style is that "everything is an object." I want to unpack this shorthand a little bit by observing that in Smalltalk-80 every value is an object; for example, the number 12 is an object; every class is an object; the program is an object; the compiler is an object; the call stack is an object; the programming environment is an object; every window is an object; and even most parts of the hardware (display, keyboard, mouse) are objects. And furthermore, every control structure is message send. Smalltalk does not have built-in control structures for constructs like `if` and `while`; instead, `if` and `while` are coded using message sends and continuation-passing style. While μSmalltalk does not go as far as enabling you to treat the interpreter and the hardware as objects, it does ensure that every value and class is an object, and it does implement every control structure using message send.

- The Learning Research Group were trying to "amplify human reach," and their efforts reached into every aspect of computer systems: hardware, compiler, language, libraries, and user interface. This book is about languages, and Smalltalk is a great one, but if you want to understand Smalltalk, looking at the language alone gives too narrow a view. In particular, Smalltalk-80 ships with a huge hierarchy of predefined classes. The basic reference on the Smalltalk-80 language, by Goldberg and Robson (1983), devotes only 90 pages to the language, but 300 pages to the predefined classes. To program *effectively* in Smalltalk, you must learn about the predefined classes and how to inherit from them.

In μSmalltalk, I also provide a set of predefined classes. By the standards of Smalltalk-80, this set is relatively small, but by the standards of this book, it is quite large. To help you focus your attention on the topics that matter most to you, I split the presentation of the μSmalltalk's predefined classes into three parts: Section 10.3 presents the predefined classes and their protocols; Section 10.5 presents the *implementations* of the predefined classes; and finally Section 10.6.6 presents the three classes that are built into the interpreter. Together, Sections 10.3 and 10.5 devote about 35 pages to the classes that are written in μSmalltalk; to the language and its semantics, I devote only about 10 pages.

10.1 Introduction to messages, classes, and inheritance

More closely than any other language in this book, μSmalltalk resembles Impcore:

- There are no static types.
- Idiomatic code is imperative, using `set` and `begin`.
- Functions are not values; anything that looks like a function call always uses a name.

In Smalltalk, there are no functions and no function calls, only methods and message sends. But because message sends play the same role that function calls play in other languages, in the concrete syntax I've chosen to make them look the same. Just as in Impcore, calling a function requires you to name the function, in Smalltalk, sending a message requires you to name the message. For example, we can send a message to the number 12 asking it if it is an object:

399 ⟨*transcript* 399⟩≡ 400▷

```
-> (isKindOf: 12 Object)
<True>
```

In this example, `isKindOf:` is the *message name*, the number 12 is the *receiver*, and the message takes class `Object` as one *argument*. But the code should remind you of Impcore. The big difference is one we won't see right away: in Smalltalk, when we see a message send, we don't always know what code is going to be executed; that decision is made dynamically. In Impcore, by contrast, the code that any function name refers to is always known statically.

The message `isKindOf:` and the objects `12` and `Object` are built in to μSmalltalk. To show some messages and objects that are not built in, we're going back to the 1970s.

In the 1970s, when Smalltalk was being created, large corporations handled accounting and other financial tasks with the aid of large "mainframe" computers, usually made by IBM. The rule was "one organization; one computer." For example, in 1975, Princeton University acquired an IBM System/370 Model 158. This behemoth occupied several large cabinets, ran at a clock rate of about 8.7 MHz, and offered programmers 4MB of main memory. And it served an entire university campus: several hundred administrators, scientists, engineers, and other faculty, plus four thousand students. WHAT DID IT COST? Against this backdrop, the Learning Research Group demonstrated that financial software could run effectively on a Xerox Alto, a 5.9 MHz machine with 128KB main memory that was designed to serve a single user. As a tutorial, they developed an example called "financial history," which you can think of as a distant ancestor to Quicken or GNU Cash: a financial history keeps track of income and outgo in named categories (Goldberg and Robson 1983). I have adapted this tutorial for μSmalltalk.

`Object` 8

Here is a short transcript that uses financial history:

400 ⟨*transcript* 399⟩+≡ ◁399 401▷

```
-> (use finance.smt)
-> (val account (initialBalance: FinancialHistory 1000))
-> (deposit:from: account 400 #salary)
-> (withdraw:for: account 50 #plumber)
-> (cashOnHand account)
1350
```

As in our other interpreters, the `use` definition loads file `finance.smt`, which in this case contains the code from Figures 10.2 and 10.3. Evaluation then proceeds as follows:

- The `val` definition sends the message `initialBalance:` to class `FinancialHistory`, with an argument of 1000. The class responds with a new *object* which is a fresh instance of `FinancialHistory` and which contains an initial balance of `1000`. That object is then stored in global variable `account`.
- The next two lines send messages to the account: we withdraw $50 for a plumber and deposit $400 salary.[2] The literals `#plumber` and `#salary` are *symbols*, just like the Scheme symbols `'plumber` and `'salary`.
- The last line sends the message `cashOnHand` to the account, which responds that it now has $1,350 available.

The code looks a lot like code in any imperative languages, but the names `initialBalance:`, `withdraw:for:`, `deposit:from:`, and `cashOnHand` name *messages*, not functions. And please look closely at the names of these messages:

- The name `cashOnHand` doesn't have any colons in it, and the message send doesn't have any arguments—just the receiver, `account`. (In Smalltalk, an "argument" is a value that is sent *in addition to* the receiver; the receiver itself is not considered an "argument.")
- The name `initialBalance:` has exactly one colon in it, and the message send has exactly one argument, the number 1000.
- The names `withdraw:for:` and `deposit:from:` have two colons apiece, and each of these message sends has two arguments.

cashOnHand 402
deposit:from: 402
FinancialHistory 402
withdraw:for: 402

These correspondences are required: in Smalltalk, the number of colons in a message's name must equal the number of arguments that sent with the message. This little convention is a master stroke of language design; even in 2012, it's quite hard to develop a static, polymorphic type system that supports flexible, dynamic programming in the Smalltalk tradition, but this syntactic requirement performs one of the most essential jobs of a static type system—making sure each method receives exactly the right number of arguments. And in proper Smalltalk syntax, the colons break the name into parts, and the parts are *interleaved* among the arguments. This convention, while unusual, leads to code that I find very easy to read (Section 10.7).

[2] For a 1970s example, we use 1970s prices.

Instance protocol for `FinancialHistory`:

`deposit:from: aNumber aSymbol`	The receiver records that `aNumber` dollars were deposited from the source named by `aSymbol`. `FinancialHistory` is a mutable abstraction, and the receiver changes itself to reflect the results of the transaction.
`withdraw:for: aNumber aSymbol`	The receiver records that `aNumber` dollars were withdrawn for the purpose named by `aSymbol`.
`cashOnHand`	The receiver answers the total cash on hand; the message expects no arguments.
`totalDepositedFrom: aSymbol`	The receiver answers the total amount deposited from the source named by `aSymbol`.
`totalWithdrawnFor: aSymbol`	The receiver answers the total amount withdrawn for the purpose named by `aSymbol`.

Class protocol for `FinancialHistory`:

`initialBalance: aNumber`	The class answers a new instance of itself. The new instance has an empty history and a balance of `aNumber`.

Figure 10.1: Protocol for `FinancialHistory`

The little transcript illustrates most of the messages that can be sent to `account`; the full set of messages (i.e., the *protocol*) is shown in Figure 10.1.[3] Messages `deposit:from:` and `withdraw:for:` are *mutators*; they tell the receiver to change its internal state. Messages `cashOnHand`, `totalDepositedFrom`, and `totalWithdrawnFor` are *observers*: they extract information from a financial history, but they don't change its state. Finally, to get a financial history to play with, you have to send a message to the *class*; the correct message is `initialBalance:`, which is a *creator*.

For completeness, here is a little more code, which shows one of the observers:

401 ⟨*transcript* 399⟩+≡ ◁400 405a▷

```
-> (withdraw:for: account 25 #plumber)
-> (deposit:from: account 5 #lemonade-stand)
-> (deposit:from: account 400 #salary)
-> (totalWithdrawnFor: account #plumber)
75
-> (totalDepositedFrom: account #salary)
800
```

account 400
deposit:from: 402
totalDeposited-From: 402
totalWithdrawn-For: 402
withdraw:for: 402

[3] If you are a Smalltalk aficionado, you will notice that the names of the messages are not the same as the names used in the Blue Book. I wanted to use different words to describe different acts: receiving a Smalltalk message is not the same as receiving money.

402 ⟨*finance classes* 402⟩≡ 406▷

```
(class FinancialHistory Object
    (cashOnHand incomes outgoes)
    (method setInitialBalance: (amount) ; private
        (set cashOnHand amount)
        (set incomes      (new Dictionary))
        (set outgoes (new Dictionary))
        self)
    (method deposit:from: (amount source)
        (at:put: incomes source (+ (totalDepositedFrom: self source) amount))
        (set cashOnHand (+ cashOnHand amount)))
    (method withdraw:for: (amount reason)
        (at:put: outgoes reason (+ (totalWithdrawnFor: self reason) amount))
        (set cashOnHand (- cashOnHand amount)))
    (method cashOnHand () cashOnHand)
    (method totalDepositedFrom: (source)
        (ifTrue:ifFalse: (includesKey: incomes source)
            [(at: incomes source)]
            [0]))
    (method totalWithdrawnFor: (reason)
        (ifTrue:ifFalse: (includesKey: outgoes reason)
            [(at: outgoes reason)]
            [0]))
    (class-method initialBalance: (amount)
        (setInitialBalance: (new self) amount))
)
```

Figure 10.2: Class `FinancialHistory`

This protocol is implemented by the definition of class `FinancialHistory`, which is shown in Figure 10.2. A class definition gives the class's name, superclass, "instance variables," and methods. (Except for a few primitive objects, each Smalltalk object is represented by a set of named variables, each of which refers to a mutable location. A location may contain any object. Because every instance of the class gets its own set of named locations, the variables are called *instance variables.*) Here's how these elements are used in the definition of `FinancialHistory`, and what they mean:

- The new class, `FinancialHistory`, is a subclass of `Object`, so it inherits all the predefined methods of `Object`.
- A `FinancialHistory` object has three instance variables: `cashOnHand`, `incomes`, and `outgoes`. Variable `cashOnHand` is an integer; `incomes` and `outgoes` are objects of the class `Dictionary`. Class `Dictionary` is defined below, but you can think of its objects as symbol tables or association lists.
- Unlike a Clu cluster, a Smalltalk class has no export list. Smalltalk always hides the *representation* of a class, but it never hides any of the *operations* on a class. If some operations are supposed to be private, this constraint is purely a matter of programming convention; it is not enforced by the language. In the `FinancialHistory` class, method `setInitialBalance` is intended to be private, as shown in a comment next to its definition.

at:,
in class Catalan 821
in class Keyed-Collection 452d
at:put:,
in class Catalan 821
in class Dictionary 453c
in class Keyed-Collection 452a
in class List 456d
ifTrue:ifFalse:,
in class Boolean 446
in class False 447a
in class False 447c
in class True 447b
includesKey: 452d
Object 8

- For consistency with Smalltalk terminology, μSmalltalk uses the keyword `method`, not `define`, to introduce an operation of a class. In addition to its arguments, each method has access to the *receiver* of the message, which is an object of the method's class (or, as we see below, of a subclass). The receiver is always referred to by the special name `self`; to refer to any of the receiver's instance variables, we use the name of the instance variable, and the reference is implicitly to the receiver.[4]

- In addition to an ordinary method, which is properly speaking an "instance method," we also define two *class methods.* Just as an instance method says what code is executed when a message is sent to an instance, a class method says what code is executed when a message is sent to the class itself. (As in full Smalltalk-80, everything in μSmalltalk is an object, and so the class `FinancialHistory` is itself represented as an object, and code can send messages to it. For example, sending the `initialBalance:` message to the class object tells it to create a new instance of `FinancialHistory` with the given amount as the initial balance.)

With the definition in hand, we can revisit some of the programming examples in the ⟨*transcript* 399⟩.

- In `(initialBalance: FinancialHistory 1000)`, the message `initialBalance:` is sent to the class. The class responds by executing the class method with the same name.

  The implementation sends `new` to `self`. In a class method, `self` stands for the class, not for an instance. The class, like all classes, inherits the `new` method from class `Class`; the `new` method answers a new, uninitialized object that is an instance of class `FinancialHistory`.[5] The class method `initialBalance:` finishes by initializing the new object, using the "private" `setInitialBalance:` method. This pattern is common: a Smalltalk object is always created by sending some message to its *class*, and it's common for the class method to initialize the object using a private instance method.

- The `setInitialBalance:` message is handled by the method of the same name. We say that the message *dispatches* to the method. According to the rules of Smalltalk, the `setInitialBalance:` method has access to two objects: the receiver (`self`), which is an instance of `FinancialHistory`, and the argument (`amount`), which is a number. The `setInitialBalance:` method also has access to the instance variables of `self`, which it assigns to by name. Instance variable `cashOnHand` is set to `amount`, and variables `incomes` and `outgoes` are set to new, empty, `Dictionary` objects. Setting these variables implicitly modifies the `self` object, which is returned.

[4] Many object-oriented languages leave the receiver implicit, but in some languages the receiver is called `this`, not `self`.

[5] Smalltalk uses the verb "to answer" in a way that is not standard English. In standard English, when you answer something, the thing answered is a request: you might "answer" an email or "answer" your phone. But in Smalltalk jargon, "answer" means "to send as the answer." A method might "answer a new object" or "answer a number." This usage should grate on your ear, but like any other jargon, it will soon seem normal.

- Method `deposit:from:` has two arguments. To account for the new deposit, the method must change two of the receiver's instance variables:

  1. It changes `incomes` by sending `incomes` the `at:put:` message of class `Dictionary`. This message is analogous to `bind` in μScheme, except that `Dictionary` is a mutable type: sending `at:put:` changes the internal state of the `Dictionary`. The value associated with `source` in `incomes` is changed to reflect the new amount received. The total previously received from this source is determined by sending the `totalDepositedFrom:` message to `self`, and then the new amount is added to it.
  2. It adds `amount` to `cashOnHand`.

- Method `withdraw:for:` is similar to `deposit:from:`.

- Method `cashOnHand` just returns the value of the instance variable `cashOnHand`. In Smalltalk, the names of methods and the names of variables are looked up in different environments (just like Impcore's functions and variables), so *method* `cashOnHand` need not be related to *variable* `cashOnHand`. But if all a method does is return the value of an instance variable, it is a good programming convention to give the method the same name as the instance variable. (Another good convention is that a method whose only effect is to mutate an instance variable should be given that variable's name with a colon added.)

- Methods `totalDepositedFrom:` and `totalWithdrawnFor:` use the `at:` message of class `Dictionary`, which is analogous to `find` in Scheme. Before sending `at:`, however, they have to check to see if the `source` or `reason` is included in the relevant dictionary. These checks are implemented using Smalltalk's model that "all control constructs are message sends."

  A check is implemented by sending the `ifTrue:ifFalse:` message to a Boolean object. The code for the true and false branches is protected by putting it in square brackets. Putting code in square brackets makes a *block* object, as explained in Section 10.3.3. The two block objects are sent to the Boolean object as arguments. If the Boolean object is `true`, it executes the code in the first block; if it is `false`, it executes the code in the second block.

We can use the class `FinancialHistory` to explore the distinction between objects and classes: every class is an object, but not every object is a class. We ask about "is-a" relationships using the `isKindOf:` message. This message is part of the instance protocol for class `Object`, so all objects respond to it.[6]

405a ⟨*transcript* 399⟩+≡ ◁401 405b▷

```
-> FinancialHistory
<class FinancialHistory>
-> (new FinancialHistory)
<FinancialHistory>
-> (isKindOf: FinancialHistory Class)
<True>
-> (isKindOf: (new FinancialHistory) Class)
<False>
-> (isKindOf: FinancialHistory Object)
<True>
-> (isKindOf: (new FinancialHistory) Object)
<True>
```

We can also use `FinancialHistory` for its intended purpose:

405b ⟨*transcript* 399⟩+≡ ◁405a 407▷

```
-> (val myaccount (initialBalance: FinancialHistory 1000))
-> (withdraw:for: myaccount 50 #insurance)
-> (deposit:from: myaccount 200 #salary)
-> (cashOnHand myaccount)
1150
-> (withdraw:for: myaccount 100 #books)
-> (cashOnHand myaccount)
1050
```

As mentioned above, a class can have *subclasses* which inherit the structure of the class. A subclass can be used to refine a class for more specific uses. A subclass inherits methods from its superclass, and inheritance is an effective way of reusing old code (the superclass) by modifying it for new circumstances (the subclass).

Figure 10.3 defines a subclass of `FinancialHistory` called `DeductibleHistory`. Its objects are financial histories which also record tax deductions. The `DeductibleHistory` class elaborates or refines the concept of financial histories as defined by `FinancialHistory`.

- The first line of the definition of `DeductibleHistory` says that `DeductibleHistory` is a subclass of `FinancialHistory`. `DeductibleHistory` therefore inherits instance variables and methods from `FinancialHistory`. (Details follow).
- `DeductibleHistory` has a new instance variable `deductible`, which contains the total of deductible expenses. However, this is not the only instance variable used to represent a `DeductibleHistory` object. Because `DeductibleHistory` is a subclass of `FinancialHistory`, a `DeductibleHistory` object also has all the instance variables associated with `FinancialHistory`. This accumulation is what is meant by inheriting instance variables.

[6] In the example above, using `(new FinancialHistory)` is dodgy: when message `new` is sent to class `FinancialHistory`, the class answers with an *uninitialized* `FinancialHistory` object.

406 ⟨*finance classes* 402⟩+≡ ◁402

```
(class DeductibleHistory FinancialHistory
    (deductible)
    (class-method initialBalance: (amount)
        (initDeductibleHistory (initialBalance: super amount)))
    (method initDeductibleHistory ()
        (set deductible 0)
        self)
    (method spend:deduct: (amount reason)
        (withdraw:for: self amount reason)
        (set deductible (+ deductible amount)))
    (method withdraw:for:deduct: (amount reason deduction)
        (withdraw:for: self amount reason)
        (set deductible (+ deductible deduction)))
    (method totalDeductions () deductible))
```

Figure 10.3: Class `DeductibleHistory`

- In addition to its own messages, a `DeductibleHistory` object can respond to any message defined in `FinancialHistory`. Since, as just explained, every instance variable of `FinancialHistory` is also an instance variable of `DeductibleHistory`, there is no trouble in sending `FinancialHistory` messages to `DeductibleHistory` objects; such messages are dispatched to the methods defined by `FinancialHistory`. This behavior is what is meant by inheriting methods.
- The `DeductibleHistory` *class* can respond to any message defined as a class method in `FinancialHistory`. But `DeductibleHistory` defines its *own* response to `initialBalance:`, which first calls the superclass method to initialize the superclass instance variables (by sending `initialBalance:` to `super`), then calls `initDeductibleHistory` to initialize its own instance variable.
- A `DeductibleHistory` object responds to three messages that a `FinancialHistory` object cannot respond to: the new messages `spend:deduct:`, `withdraw:for:deduct:`, and `totalDeductions`. The implementations of methods `spend:deduct:` and `withdraw:for:deduct:` illustrate another form of reuse; they send `self` the message `withdraw:for:`, which is defined in `FinancialHistory`.

`FinancialHistory`
402
`withdraw:for:`
402

A DeductibleHistory object responds to messages defined either in DeductibleHistory or in FinancialHistory:

407 ⟨*transcript* 399⟩+≡ ◁405b 411a▷

```
-> (val myaccount (initialBalance: DeductibleHistory 1000))
-> (withdraw:for: myaccount 50 #insurance)
-> (deposit:from: myaccount 200 #salary)
-> (cashOnHand myaccount)
1150
-> (totalDeductions myaccount)
0
-> (spend:deduct: myaccount 100 #mortgage)
-> (cashOnHand myaccount)
1050
-> (totalDeductions myaccount)
100
-> (withdraw:for:deduct: myaccount 100 #3-martini-lunch 50)
-> (cashOnHand myaccount)
950
-> (totalDeductions myaccount)
150
```

With these two kinds of financial history under our belts, we are ready to look at the full μSmalltalk language, the protocols of the predefined classes, and an extended example. After those sections, the rest of the chapter tackles the implementations of the predefined classes and of the language.

10.2 The μSmalltalk language

10.2.1 Concrete syntax

The expressions of μSmalltalk are like a combination of expressions from Impcore and μScheme. All three languages have variables, `set`, and `begin`. Instead of function application, μSmalltalk has message send. Like Impcore's functions, μSmalltalk's messages must be referred to by name.[7]

[7] Impcore's functions are not values, but μScheme's functions *are* values. Are μSmalltalk's functions values? The question is ill posed: Smalltalk has no functions. Smalltalk has only objects, to which messages can be sent.

In addition to message send and to the forms that are familiar from Impcore and μScheme, a μSmalltalk expression can be a *block*, which is a bit like a `lambda` abstraction in Scheme. A block specifies zero or more formal parameters and a body. The body contains a sequence of expressions, which are executed in, well, sequence. Because parameterless blocks are used frequently to write control flow, μSmalltalk provides a special syntax for a parameterless block: the μSmalltalk expression `[`$\texttt{exp}_1 \ldots \texttt{exp}_n$`]` is syntactic sugar for the expression `(block () `$\texttt{exp}_1 \ldots \texttt{exp}_n$`)`. This syntax is borrowed from Smalltalk-80.

exp ::= *literal*
| *variable-name*
| `(set` *variable-name exp*`)`
| `(begin` {exp}`)`
| `(`*message-name exp* {*exp*}`)`
| `(block (`{argument-name}`)` {*exp*}`)`
| `[`{*exp*}`]`

Every *message-name* has an *arity*, which is the number of arguments that the message expects. If the *message-name* begins with a nonletter, its arity is 2. If the *message-name* begins with a letter, its arity is the number of colons in the name, *except* if *message-name* is one of the special names `if`, `while`, or `value`. The name `if` has arity 2, `while` has arity 1, and `value` may have any arity.

Like μScheme, μSmalltalk has integer and symbol literals. It has array literals instead of list literals, and a literal is introduced with `#`, not with a quote.

literal ::= *integer-literal*
| `#`*name*
| `#(`{array-element}`)`

array-element
::= *integer-literal*
| *name*
| `(`{array-element}`)`

Definitions include the usual `val`, *exp*, `define`, and `use` that we see in Impcore and μScheme, as well as a class definition. But in μSmalltalk, `define` defines a block, not a function.

def ::= `(val` *variable-name exp*`)`
| *exp*
| `(define` *block-name* `(`*formals*`)` *exp*`)`
| `(class` *subclass-name superclass-name*
({*instance-variable-name*}) {*method-definition*}`)`
| `(use` *file-name*`)`

method-definition
::= `(` (`method` | `class-method`) *method-name* `(`*formals*`)` [`(locals` *locals*`)`] {*exp*}`)`
| `(` (`method` | `class-method`) *method-name* `primitive` *primitive-name*`)`

formals ::= {formal-parameter-name}

locals ::= {local-variable-name}

A method may be specified by giving its formal parameters and body, or it may be a primitive method specified by name.

10.2.2 Values

In μSmalltalk, all values are objects. As in Smalltalk-80, one has to know the predefined objects. We describe the interfaces to these objects in Section 10.3 and the implementations in Section 10.5, but a short preview may be helpful.

Properties shared by all values

Every value is an object, and every object is an instance of some class. Class `Object` is the root of the class hierarchy, so every other class inherits from it. This means that every object responds to messages defined on instances of class `Object`. These messages include `isKindOf:`, `=`, `print`, and a number of others. They are a bit like predefined functions in other languages, with an important difference: you can redefine them.

The undefined object, `nil`

The value `nil` is the sole instance of class `UndefinedObject`. It is used, by convention, to represent a "bad" or "undefined" value. This usage is quite different from the use of *nil* in Scheme, where it represents the empty list.

Other predefined objects

A literal symbol, written `#`*name*, is a value of class `Symbol`. Like all objects, symbols can usefully be printed and compared for equality.

A Boolean object responds to the messages `ifTrue:ifFalse:` and `whileTrue:`. The arguments for these messages are *blocks*, which are executed or not depending on the value of the Boolean. The Boolean objects `true` and `false` are predefined. For consistency with other languages in this book, `ifTrue:ifFalse:` and `whileTrue:` may be abbreviated as `if` and `while`.

Classes are objects

Every class definition creates a class object, which itself has a class that is a subclass of class `Class`. A class object can respond to `new` and possibly to other class-dependent initialization messages. In μSmalltalk, a class object also responds to the message `protocol`, which prints all the messages that a class and its instances know how to respond to.

Every object is an instance of a class, so even when an object represents a class, it is also an instance of a class. The object representing class `C` is an instance of `C`'s *metaclass*, and in fact it is the *only* instance of `C`'s metaclass. If an object `c` is an instance of class `C`, the methods defined on `C` determine what messages `c` can respond to. Similarly, the methods defined on `C`'s metaclass determine what messages `C` can respond to. In other words, the metaclass provides a place to put `C`'s class methods. Because every metaclass inherits from class `Class`, every class object responds to the `new` message, which is defined on class `Class`.

In μSmalltalk, a metaclass is not a value; it is simply a convenient placeholder for class methods. In Smalltalk-80, every object responds to the `class` message, which gives the class of an object, the metaclass of a class, and so on. Furthermore, the metaclass objects make it possible to learn the names of methods and instance variables, to execute methods by name, and more. These kinds of *reflective* facilities, with which a language can manipulate its own objects, are powerful but hard to implement. And to use them effectively requires a complete grasp of all the details of the language and its semantics.

10.2.3 Names

Just as Impcore has distinct name spaces for functions and values, μSmalltalk has distinct name spaces for messages and variables. The name of a variable is resolved *statically*; that is, we can tell by looking at a class definition what each variable name in each method must stand for. The name of a message is resolved *dynamically*; that is, we can't tell until a message is sent what method will be executed to respond to it.

The name of a variable stands for one of the following:

1. A formal parameter of a block in which it appears
2. A local variable of a method in which it appears
3. A formal parameter of a method in which it appears
4. An instance variable of an object which is an instance of the class in the definition of which the variable appears
5. A global variable

To tell which of these things a name stands for, look at the source code where the name appears. For example, a name in a top-level expression doesn't appear inside a method or a class definition, so it stands either for a parameter of a block or for a global variable.

By rule 4, the instance variables of an object of class `C` are those named in the definition of `C` *or in any of its ancestors.* (Name conflicts are best avoided.)

The rules for the meanings of names enforce *information hiding: only a method executed on an object has direct access to the instance variables of that object.* In other words, an object's instance variables are *private*; not even another object of the same class can get access to them. In effect, all objects communicate with other objects as if they were purely abstract. (See also Exercise 29 on page 514, part a.)

The names `self` and `super` are not, properly speaking, variables; for one thing, they cannot be set. Within a method, `self` stands for the object that received the message which caused the method to be executed. The name `super` stands for the same object, but with different rules for method dispatch.

To send a message `m` to an ordinary object of class `C`, look in `C`'s definition for a method named `m`. If that fails, look in the definition of `C`'s superclass, and so on. When you find a definition of a method for `m`, execute that method. If you reach the top of the hierarchy without finding a definition of `m`, an error has occurred. In general, we can't tell by looking at the source code what the class of a receiver will be, so we also can't tell what method a message name will stand for at run time.

Sending to `super` changes the rules. To send a message `m` to `super`, we start the message search not in class `C` but in the superclass of the class where `super` appears in the source code. By examining the source code, we can tell what method will be executed.

10.2.4 Message sends, inheritance, and method dispatch

An object of class `C` can respond to any message for which a method is defined in an ancestor of `C`. While it is useless to redefine an instance variable, it can be quite useful for `C` to redefine a method. In that case, when the relevant message is sent to an object of class `C`, the message is dispatched to `C`'s definition of the method, which is executed. The way the rules play out might surprise you, so let's look at a contrived example.

Suppose classes B (the superclass or ancestor) and C (the subclass) are defined as follows:

411a ⟨*transcript* 399⟩+≡ ◁407 411b▷

```
-> (class B Object ()
     (method m1 () (m2 self))
     (method m2 () #B))
-> (class C B ()
     (method m2 () #C))
```

What happens when message m1 or m2 is sent to an object of class B or C? When m1 or m2 is sent to an object of class B, the answer is the symbol B. And when m2 is sent to an object of class C, method dispatch finds the definition of m2 within C, and the result is symbol C. These cases are easy. The potentially surprising case occurs when m1 is sent to an object of class C:

411b ⟨*transcript* 399⟩+≡ ◁411a 412▷

```
-> (val x (new C))
-> (m1 x)
C
```

Here is what happens:

1. When m1 is sent to x, x is an instance of class C, but class C does not define method m1. But a search reveals that m1 is defined by C's superclass, class B. The message is dispatched to B's m1 method, which executes.

2. B's m1 method sends message m2 to self—which is to say, to x. The search for method m2 begins in x's class, namely C.

3. A definition of m2 is found in C, and the message send dispatches to it. Executing class C's m2 method answers the symbol C.

This example illustrates a crucial fact about Smalltalk: *the search for method always begins in the class of the receiver.* Many more examples of method dispatch appear below.

Does message dispatch *always* begin in the class of the receiver? Well, almost always. There is a special exception for sending a message to **super**. The message is sent to **self**, but the method search starts *in the superclass of the class in which the message to* **super** *appears.* That is, unlike the starting place for a normal message send, the starting place for a message to **super** is *statically determined*, not dependent on the class of the receiver **self**. This behavior is useful when one wants to guarantee to execute a particular method from the superclass.

By far the most common use of **super** is in class methods that initialize objects, like **new**. *Every* class has a **new** method. A properly designed **new** method not only allocates a new object but also establishes the private invariants of its class. (See, for example, Exercise 8 on page 509.) Simply sending **new** to **self** executes only the **new** method defined on the class of the object being created. But if there are invariants associated with the superclass, those invariants need to be established too. The idiomatic way to establish *all* the invariants is for each subclass to send **new** to **super**, establishing the superclass invariants, then execute additional code to establish any subclass invariants. By "chaining together" **new** methods in this way, we establish the invariants of every superclass. (If a subclass has no invariants of its own, it can take a shortcut and simply inherit **new**.)

Object B

`Object` instance protocol

`isKindOf: aClass`	Answer whether the argument, `aClass`, is a superclass or class of the receiver.
`isMemberOf: aClass`	Answer whether the argument, `aClass`, is exactly the receiver's class.
`= anObject`	Answer whether the argument should be considered equal to the receiver.
`!= anObject`	Answer whether the argument should be considered not equal to the receiver.
`isNil`	Answer whether the receiver is `nil`.
`notNil`	Answer whether the receiver is not `nil`.
`print`	Print the receiver on standard output.
`println`	Print the receiver, then a newline, on standard output.
`error: aSymbol`	Issue a run-time error message which includes `aSymbol`.
`subclassResponsibility`	Report to the user that a method specified in the superclass of the receiver should have been implemented in the receiver's class.

Figure 10.4: Protocol for instances of class `Object`

10.3 The initial basis

Smalltalk may be the simplest language used by real programmers; it is so simple that μSmalltalk models almost all of it. Both full Smalltalk-80 and μSmalltalk provide assignment, message send, block creation, and class definition; μSmalltalk lacks only a feature called *nonlocal return*, which simplifies the efficient implementation of searching methods. How can such a simple language be useful for real problems? Because the power of the system is not in the *language*, but in the predefined objects—the *initial basis*. Most of these objects are, of course, classes.

The initial basis of μSmalltalk is far smaller than that of full Smalltalk, but it is still quite large, and it suffices to give much of the flavor of Smalltalk programming. This section describes the predefined classes from the top down.

10.3.1 Protocol for all objects

Every μSmalltalk object responds to the messages in Figure 10.4, which are defined on the predefined class `Object`. Messages `isKindOf:` and `isMemberOf:` make it possible to test for the class of a receiver. For example, this session tells us that `3` is some subclass of `Number`, but that the symbol `#3` is not a number at all.

Number 459b

412 ⟨*transcript* 399⟩+≡ ◁411b 413▷

```
-> (isKindOf: 3 Number)
<True>
-> (isMemberOf: 3 Number)
<False>
-> (isKindOf: #3 Number)
<False>
```

`new`	The receiver is a class; answer a new instance of that class. A class may override `new`, e.g., if it needs arguments to initialize a newly created instance.
`protocol`	The receiver is a class; print the class messages as well as all the messages an instance of that class knows how to respond to.
`localProtocol`	The receiver is a class; print only those class and instance messages that are defined by this class, not messages whose implementations are inherited from the superclass.

Figure 10.5: Protocol for classes

Methods `=` and `!=` test for equality and inequality, while `isNil` and `notNil` test for equality and inequality with `nil`.

The `print` method is used by the μSmalltalk interpreter itself to print the values of objects; `println` follows with a newline.

The `error:` and `subclassResponsibility` messages are used to report errors.

10.3.2 Protocol for classes

Because even a class is an object, a class itself can answer messages. Every class object inherits from `Class`, so it responds to the protocol in Figure 10.5. Using `class-method`, you can define additional methods on a class object.

10.3.3 Blocks and Booleans

A block is Smalltalk's way of delaying the evaluation of an expression. Writing a block creates an object with an expression inside; the expression is evaluated when the `value` message is sent to the block.

413 ⟨*transcript* 399⟩+≡ ◁412 414a▷

```
-> (val index 0)
-> [(set index (+ index 1))]
<Block>
-> index
0
-> (value [(set index (+ index 1))])
1
-> index
1
```

A block is much like a `lambda` expression in Scheme, but because Smalltalk has no functions, a block is an object, and it is evaluated not by applying it, but by sending it the `value` message.

`value` 452b

Just like any object, a block can be assigned to a variable and used later.

414a ⟨*transcript* 399⟩+≡ ◁413 414b▷

```
-> (val incrementBlock [(set index (+ index 1))])
<Block>
-> (val sumPlusIndexSquaredBlock [(+ sum (* index index))])
<Block>
-> (val sum 0)
0
-> (set sum (value sumPlusIndexSquaredBlock))
1
-> (value incrementBlock)
2
-> (set sum (value sumPlusIndexSquaredBlock))
5
```

What is most surprising about blocks is not their semantics but the way they are used. Smalltalk uses blocks to implement conditional expressions, much as the pure λ-calculus uses functions to implement conditional expressions. In other languages, including Scheme, an expression of the form `(if e1 e2 e3)` is treated specially by the interpreter, which arranges to evaluate only `e2` or `e3` as needed. In μSmalltalk, as in full Smalltalk-80, `if` is not built in; message passing is the *only* control structure. The `if` expression above sends message `if` to receiver `e1`, with arguments `e2` and `e3`, both of which are evaluated. To make `if` behave as expected, the right thing to do is use blocks to delay the evaluation of `e2` and `e3`, by writing `(if e1 [e2] [e3])`.[8] This transcript illustrates the difference:

414b ⟨*transcript* 399⟩+≡ ◁414a 414c▷

```
-> (if (< sum 0) [#negative] [#positive])
positive
-> (if (< sum 0)  #negative   #positive )
run-time error: Symbol does not understand message value
Method-stack traceback:
  Sent 'value' in initial basis, line 25
  Sent 'ifTrue:ifFalse:' in initial basis, line 19
  Sent 'if' in standard input, line 98
```

On the first line, `(< sum 0)` produced the Boolean object `false`. Sending `if` to `false` resulted in sending `ifTrue:ifFalse:` to `false`, which in turn sent `value` to the block `[#positive]`, which answered the symbol `#positive`, which was the result of the entire expression. On the second line, `false` tried to send `value` to the symbol `#positive`, which resulted in an error message and a stack trace.

if 446
index 413
value 452b
while 448a

To implement `while`, not only does the body of the `while` expression have to be a block, the condition has to be a block as well, because it must be evaluated repeatedly.

⟨*transcript* 399⟩+≡ ◁414b 419a▷

```
-> (while [(< sum 10000)] [(set sum (* 5 sum)) (println sum)])
25
125
625
3125
15625
nil
```

[8] This code is another example of *continuation-passing style*, which we use in Section 3.10 to implement a simple solver. In Smalltalk, we implement control flow by passing two blocks—the continuations—to a Boolean. The value `true` continues by executing the first block; `false` continues by executing the second.

Instance protocol for Booleans:

`& aBoolean`	Answer the conjunction of the receiver and the argument.
`| aBoolean`	Answer the disjunction of the receiver and the argument.
`not`	Answer the complement of the receiver.
`eqv: aBoolean`	Answer `true` if the receiver is equivalent to the argument.
`xor: aBoolean`	Answer `true` if the receiver is different from the argument (exclusive or).
`and: alternativeBlock`	If the receiver is `true`, answer the value of the argument; otherwise, answer `false` (short-circuit conjunction).
`or: alternativeBlock`	If the receiver is `false`, answer the value of the argument; otherwise, answer `true` (short-circuit disjunction).
`ifTrue:ifFalse: trueBlock falseBlock`	If the receiver is true, evaluate `trueBlock`, otherwise evaluate `falseBlock`.
`if trueBlock falseBlock`	Abbreviation for `ifTrue:ifFalse:`
`ifTrue: trueBlock`	If the receiver is true, evaluate `trueBlock`, otherwise answer `nil`.
`ifFalse: falseBlock`	If the receiver is false, evaluate `falseBlock`, otherwise answer `nil`.

Instance protocol for blocks:

`value`	Evaluate the receiver and return its value.
`value` *arguments*	Bind *arguments* to the formal parameters of the receiver, evaluate its body, and return its value.
`while bodyBlock`	Abbreviation for `whileTrue:`
`whileTrue: bodyBlock`	Send `value` to the receiver, and if the response is true, send `value` to `bodyBlock` and repeat. When the receiver responds `false`, answer `nil`.
`whileFalse: bodyBlock`	Send `value` to the receiver, and if the response is false, send `value` to `bodyBlock` and repeat. When the receiver responds `true`, answer `nil`.

Figure 10.6: Protocols for Booleans and blocks

The interpreter prints `nil` because the value of the `while` expression is `nil`.

Figure 10.6 shows the protocols for blocks and Booleans.

10.3.4 Collections

In this section, we explore a sub-hierarchy containing a variety of useful classes, including `Dictionary`. Classes in this sub-hierarchy are those that *contain* things, in a sense made precise below. At the top of the sub-hierarchy is the class `Collection`, which is a subclass of `Object`. Collections include sets, lists, and arrays, as well as dictionaries.

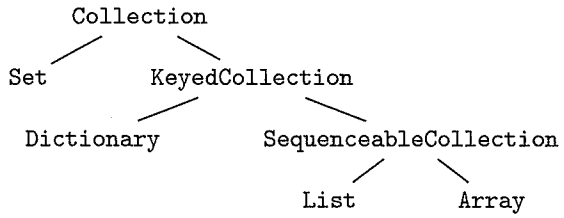
Smalltalk-80 also has a Collection sub-hierarchy, including similar classes, though structured somewhat differently; it is discussed in section 10.7.2. My version is inspired by Budd's (1987) Collection hierarchy.

What does it mean to say that `Collections` contain things? Above all, it means that any `Collection` object responds to the message `do:`. This message must be defined so that, by sending (`do:` *collection* (`block (x)` *body*)), we evaluate *body* once for each item `x` in *collection*. The `do:` method acts very much like a Clu iterator, but thanks to the power of Smalltalk blocks, there is no need for a special language construct; `do:` is a method like any other.

Collections are mutable; a typical `Collection` object should respond to message `add:` by adding the argument to itself, and to the message `remove:` by removing the argument from itself. Some collections, like `Arrays`, have a fixed number of elements; when receiving `add:` or `remove:` messages, such collections report errors.

We speak of "`Collection` objects," but `Collection` itself is an *abstract class*, meaning it has no useful instances of its own. (Like the idea of a private method, the idea of an "abstract class" is not an official part of the Smalltalk language; it is a programming convention.) A client should not create an instance of an abstract class such as `Collection`; it should instead create instances of concrete subclasses. A "collection" or "`Collection` object" is any object of a subclass (or, more generally, any descendant) of `Collection`. To put it differently, a collection is any object that inherits the `Collection` protocol.

This picture shows the shape of the `Collection` hierarchy, so we can see what classes inherit the protocol.



Here is an informal description of what each class does or what it is:

`Collection`: A collection contains objects.

`Set`: A set contains objects in no particular order.

`KeyedCollection`: A keyed collection contains objects that are associated with keys, and these objects can be added or retrieved by using the key.

`Dictionary`: A dictionary is a keyed collection that assumes as little as possible about keys: in a dictionary, keys need only to be comparable for equality.

`SequenceableCollection`: A sequenceable collection is a keyed collection whose keys are consecutive integers.

`List`: A list is a sequenceable collection that can grow or shrink. Objects can be added to or removed from the begining or end of a list. Unlike a list in Scheme or ML, a Smalltalk list is *mutable*.

`Array`: An array is a sequenceable collection whose elements can be accessed in constant time—but it cannot grow or shrink.

In our implementation of this hierarchy, we use just two representations: lists and primitive arrays. An object of class `Array` is represented by a primitive array; all other collection objects have instance variables that contains lists. The classes of these objects allocate lists and send messages to them, so they are *clients* of `List`; this is not the same as inheriting from `List`.

Figure 10.7 on page 418 describes the collection protocol in detail.

- Each collection *class* responds to a `with:` message, which creates a collection containing a single element.
- The instance methods in the first group are *mutators*; they define ways of adding and removing elements.
- The instance methods in the second group are *observers*; they define ways of finding out about the elements in a collection. The `includes:` and `occurrencesOf:` observers ask about elements directly; the `detect:` and `detect:ifNone:` observers ask for *any* element satisfying a given predicate, which is represented as a block.
- The instance methods `do:` and `inject:into:` are *iterators*; they repeat a computation once for every element in the collection. The `do:` method performs a computation only for side effect, while `inject:into:` accumulates and returns a value. These two methods correspond closely to the ML functions `List.app` and `List.foldl`, but they are defined on *all* collections, not only on lists.
- The instance methods in the final group are *producers*; given a collection, such a method makes a new collection, without mutating the original collection. Methods `select:` and `collect:` correspond to the μScheme functions `filter` and `map`, which are also defined in ML. Method `asSet` returns a set containing the elements of the receiver.

`Collection` is an abstract class, so a client should not send a `new` or `with:` message directly to `Collection`, but only to one of its subclasses.

Class protocol for `Collection`:

`with: anObject`	Create and answer a singleton collection holding the given object.

Instance methods for `Collection`:

`add: newObject`	Include the argument, `newObject`, as one of the elements of the receiver. Answer `newObject`.
`addAll: aCollection`	Add every element of the argument to the receiver; answer `aCollection`.
`remove: oldObject`	Remove the argument, `oldObject`, from the receiver. If no element is equal to `oldObject`, report an error; otherwise, answer `oldObject`.
`remove:ifAbsent: oldObject exnBlock`	Remove the argument, `oldObject`, from the receiver. If no element is equal to `oldObject`, answer the result of evaluating `exnBlock`; otherwise, answer `oldObject`.
`removeAll: aCollection`	Remove every element of the argument from the receiver; answer `aCollection` or report an error.
`isEmpty`	Answer whether the receiver contains any elements.
`size`	Answer how many elements the receiver contains.
`includes: anObject`	Answer whether the receiver contains `anObject`.
`occurrencesOf: anObject`	Answer how many of the receiver's elements are equal to `anObject`.
`detect: aBlock`	Answer the first element `x` in the receiver for which `(value aBlock x)` is true, or report an error if none.
`detect:ifNone: aBlock exnBlock`	Answer the first element `x` in the receiver for which `(value aBlock x)` is true, or answer `(value exnBlock)` if none.
`do: aBlock`	For each element `x` of the collection, evaluate `(value aBlock x)`.
`inject:into: thisValue binaryBlock`	Evaluate `binaryBlock` once for each element in the receiver. The first argument of the block is an element from the receiver; the second argument is the result of the previous evaluation of the block, starting with `thisValue`. Answer the final value of the block.
`select: aBlock`	Answer a new collection like the receiver, containing every element `x` of the receiver for which `(value aBlock x)` is true.
`reject: aBlock`	Answer a new collection like the receiver, containing every element `x` of the receiver for which `(value aBlock x)` is false.
`collect: aBlock`	Answer a new collection like the receiver, containing `(value aBlock x)` for every element `x` of the receiver.
`asSet`	Answer a set containing the elements of the receiver.

Figure 10.7: Protocol for class `Collection`

The simplest subclass of `Collection` is `Set`; to get a feel for sets and for the `Collection` protocol, look at this transcript:

419a ⟨*transcript* 399⟩+≡ ◁414c 419b▷

```
-> (val s (new Set))
Set( )
-> (size s)
0
-> (add: s 2)
-> (add: s #abc)
-> (includes: s 2)
<True>
-> (add: s 2)
-> (size s)
2
-> (do: s (block (x) (println x)))
2
abc
nil
```

Remember that when a message is sent to a receiver, the method has access not only to the values of the arguments but also to the receiver. For example the definition of the `add:` method has only one argument, that being the item to be added, but it also has access to the collection to which the item is added.

Our next examples mix two kinds of collections: sets and arrays.

419b ⟨*transcript* 399⟩+≡ ◁419a 422▷

```
-> (set s (asSet #(1 2 3 1 2 3)))
Set( 1 2 3 )
-> (addAll: s #(1 2 3 a b c))
( 1 2 3 a b c )
-> (includes: s #b)
<True>
-> (size s)
6
-> (val s2 (reject: s (block (x) (isKindOf: x Number))))
Set( a b c )
```

How do we ensure that `Set` works correctly with any kind of object? In the language of Section 3.9 on page 100, how do we make `Set` polymorphic? As in Section 3.9, the essential requirement is that `Set` must use the correct definition of equality for the objects it contains. In μScheme, we use higher-order functions, arranging to pass an equality function to the set operations. But in Smalltalk, the right thing happens with almost no effort: any object stored in a `Set` should define the standard `=` method in such a way as to do the right thing. The methods of the `Set` class need only send `=` messages to the objects contained in the set, and method dispatch makes sure that the correct definition of `=` is used. Method dispatch easily gets the right definition from the right place; this way of making polymorphism easy is one of the advantages of object-oriented programming.

Instance protocol for `KeyedCollection`:

`at:put: key value`	Modify the receiver by associating `key` with `value`. May add a new value or replace an existing value.
`at: key`	Answer the value associated with `key`, or report an error if there is no such value.
`at:ifAbsent: key aBlock`	Answer the value associated with `key`, or the result of evaluating `aBlock` if there is no such value.
`includesKey: key`	Answer `true` if there is some value associated with `key`, or `false` otherwise.
`keyAtValue: value`	Answer the key associated with `value`, or if there is no such value, report an error.
`keyAtValue:ifAbsent: value aBlock`	Answer the key associated with `value`, or if there is no such value, answer the result of evaluating `aBlock`.
`associationAt: key`	Answer the `Association` in the collection with key `key`, or if there is no such `Association`, report an error.
`associationAt:ifAbsent: key exnBlock`	Answer the `Association` in the collection with key `key`, or if there is no such `Association`, answer the result of evaluating `exnBlock`.
`associationsDo: aBlock`	Iterate over all `Associations` in the collection, evaluating `aBlock` with each one.

Class protocol for `Association`:

`key:value: key value`	Create a new association with the given key and value.

Instance protocol for `Association`:

`key`	Return the receiver's key.
`value`	Return the receiver's value.
`key: key`	Set the receiver's key to `key`.
`value: value`	Set the receiver's value to `value`.

Figure 10.8: Protocols for keyed collections

Instance protocol for `SequenceableCollection`:

`first`	Answer the first element of the receiver.
`firstKey`	Answer the integer key that is associated with the first value of the receiver.
`last`	Answer the last element of the receiver.
`lastKey`	Answer the integer key that is associated with the last value of the receiver.

Figure 10.9: Protocol for sequenceable collections

Keyed collections and class `Dictionary`

`KeyedCollection` is another abstract class; an object that inherits from `KeyedCollection` represents a set of key-value pairs in which no key occurs more than once. This abstraction is one of the most frequently used abstractions in computing, and depending on context, it may be called an "associative array," a "dictionary," a "finite map," or a "table." Smalltalk uses the word "keyed collection" to encompass not only general-purpose key-value data structures but also special-purpose structures including lists and arrays.

An object of class `KeyedCollection` responds to the `Collection` protocol is if it were a simple collection of values, with no keys. But `KeyedCollection` adds additional messages to the protocol (Figure 10.8), and using these messages, you can present a key and use it to look up or change the corresponding value.

- The `at:put:` message is a new mutator; `(at:put: kc key value)` modifies `kc` by associating `value` with `key`.
- The new observers `at:` and `keyAtValue:` return the value associated with a key or the key associated with a value. The observer `at:ifAbsent:` can be used to learn whether a given key is in the collection. To learn whether a given *value* is in a collection, we use the observer `includes:` from the `Collection` protocol.
- The observer `associationAt:` and the iterator `associationsDo:` provide access to the key-value pairs directly, in the form of `Association` objects.

Among these new messages, the most frequently used are `at:` and `at:put:`. Among the old messages that `KeyedCollection` inherits from `Collection`, the most frequently used may be `size`, `isEmpty`, and `includes:`.

`KeyedCollection` is an abstract class; a client should create instances of subclasses only. The simplest subclass is called `Dictionary`; except for assuming that, like every object, its keys respond correctly to the `=` message, it makes no other assumptions about its keys. A `Dictionary` object can be represented as a list of "associations."

Sequenceable collections

The abstract class `SequenceableCollection` represents a keyed collection whose keys are consecutive integers. This class defines additional protocol, which is shown in Figure 10.9. Methods `firstKey` and `lastKey` return the first and last keys, and `first` and `last` return the first and last values.

Instance protocol for `List`:

`addLast: anObject`	Add `anObject` to the end of the receiver and answer `anObject`.
`addFirst: anObject`	Add `anObject` to the beginning of the receiver and answer `anObject`.
`removeFirst`	Remove the first object from the receiver and answer that object. Causes an error if the receiver is empty.
`removeLast`	Remove the last object from the receiver and answer that object. Causes an error if the receiver is empty.

Figure 10.10: Protocol for lists

Lists A `List` is a sequenceable collection that can change size: we can add or remove an element at either end of a list. The add and remove operations appear in the protocol shown in Figure 10.10. Unlike a Scheme list, a Smalltalk list is *mutable*: adding or removing an element mutates the original list, rather than creating a new one. The `add:` message is a synonym for `addLast:`.

Here is how `List` is used:

422 ⟨*transcript* 399⟩+≡ ◁419b 426a▷

```
-> (val l (new List))
List( )
-> (addLast: l #a)
a
-> (add: l #b)
b
-> l
List( a b )
-> (addFirst: l #z)
-> (first l)
z
-> (addFirst: l #y)
-> (at: l 3)
a
-> (removeFirst l)
y
-> l
List( z a b )
```


Arrays An array is a sequenceable collection that cannot change size, but that implements `at:` and `at:put:` in constant time. Arrays are found in many programming languages; in Smalltalk, every array is one-dimensional, and the first element's key is 1.

`species`	Answer a class that should be used to create new instances of collections like the receiver, to help with the implementation of `select:`, `collect:`, and similar methods.
`printName`	Print the name of the object's class, to help with the implementation of `print`. (Almost all `Collection` objects print as the name of the class, followed by the list of elements in parentheses. `Array` objects omit the name of the class.)

Figure 10.11: Private methods internal to `Collection` classes.

An instance of `Array` responds to the messages required by the `SequenceableCollection` protocol. (Because an array cannot change size, it responds to `add:` and `remove:` with errors.) The `Array` *class* responds to the standard messages for `SequenceableCollection` classes, and also to these messages:

`new: anInteger`	Create and answer an array of size `anInteger` in which each element is `nil`.
`from: aCollection`	Create and answer an array containing all the elements of `aCollection`. If `aCollection` is an ordered (sequenceable) collection, then the `from:` method preserves the order.

Inheriting from `Collection`

It might seem an enormous job to create a new subclass of `Collection`; after all, Figure 10.7 shows an enormous number of messages, which any `Collection` object must answer all of. But the `Collection` class is carefully structured so that a subclass need implement only *three* of the many methods in Figure 10.7: `do:`, to iterate over elements, `add:`, to add an element, and `remove:ifAbsent:`, to remove an element. In addition, a subclass of `Collection` must implement the two methods in Figure 10.11, which are private to the implementation of `Collection`. The idea of a private method, like the idea of an abstract class, is a programming convention that is not enforced by the Smalltalk language. A private method of a class should be called only by objects of that class and its subclasses, not by client objects.

Instance protocol for `Magnitude`:

`= aMagnitude`	Answer whether the receiver equals the argument.
`< aMagnitude`	Answer whether the receiver is less than the argument.
`> aMagnitude`	Answer whether the receiver is greater than the argument.
`<= aMagnitude`	Answer whether the receiver is no greater than the argument.
`>= aMagnitude`	Answer whether the receiver is no less than the argument.
`min: aMagnitude`	Answer the lesser of the receiver and `aMagnitude`.
`max: aMagnitude`	Answer the greater of the receiver and `aMagnitude`.

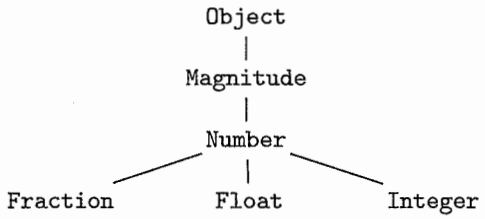
Instance protocol for `Number`:

`negated`	Answer the negation of the receiver.
`reciprocal`	Answer the reciprocal of the receiver.
`abs`	Answer the absolute value of the receiver.
`+ aNumber`	Answer the sum of the receiver and the argument.
`- aNumber`	Answer the difference of the receiver and the argument.
`* aNumber`	Answer the product of the receiver and the argument.
`/ aNumber`	Answer the quotient of the receiver and the argument. The quotient is *not* rounded to the nearest integer, so the quotient of two integers may be a different kind of number.
`negative`	Answer whether the receiver is negative.
`positive`	Answer whether the receiver is nonnegative (sic).
`strictlyPositive`	Answer whether the receiver is strictly positive.
`raisedToInteger: anInteger`	Answer the receiver, raised to the integer power `anInteger`.
`squared`	Answer the receiver squared.
`sqrtWithin: eps`	Answer a number that is within `eps` of the square root of the receiver.
`sqrt`	Answer the square root of the receiver, to within some `eps` defined by the implementation.
`coerce: aNumber`	Answer a `Number` that is of the same *kind* as the receiver, but represents the same *value* as the argument.
`asInteger`	Answer the integer nearest to the value of the receiver.
`asFraction`	Answer the fraction nearest to the value of the receiver.
`asFloat`	Answer the floating-point number nearest to the value of the receiver.

Figure 10.12: Protocols for magnitudes and numbers

10.3.5 Numbers

The collections make up a large part of μSmalltalk's initial basis. Much of the rest of the basis deals with numbers. The numeric classes illustrate a natural use of inheritance: defining arithmetic operations over different numeric representations. The numeric subhierarchy is:



Objects of class `Integer` represent integers; `Fractions` and `Floats` represent numbers with fractional parts. A `Fraction` represents a rational number by a pair of integers (a numerator and a denominator). A `Float` uses a floating-point representation, giving, as integers, a mantissa and an exponent of 10. The `Magnitude` and `Number` protocols together define all the usual arithmetic operations, which we can perform on any of the three kinds of numbers. μSmalltalk supports *homogeneous* operations, not mixed arithmetic between different kinds of numbers. (Programming with mixed arithmetic is very convenient, and Exercise 29 on page 514 asks you to implement arithmetic that works on a mix of integers and fractions.) `Magnitude` supports only operations defined on all types that can be ordered; `Number` supports additional operations, such as addition, that can be performed only on numbers. Figure 10.12 shows both protocols.

Each concrete subclass of `Magnitude` must implement the `=` and `<` methods; these two basic methods can be used to implement all the others, and the definition of `Magnitude` includes default implementations. The `Magnitude` protocol is useful for any objects that can be ordered, and it can be used to build search trees, sorted collections, etc. In Smalltalk-80, magnitudes include dates, times, and characters; in μSmalltalk, the predefined magnitudes include only numbers.

The messages in the `Number` protocol can be divided into several groups.

- A `Number` responds to the usual four binary arithmetic operations, as well as absolute value, negation and reciprocal.
- A `Number` responds to three sign-querying messages.
- A `Number` responds to messages asking for a few other mathematical operations: power, square, and square root.
- A `Number` can be converted to any other kind of `Number`.

Integers Integers provide not just the standard arithmetic operations, but also `div:`, `mod:`, `gcd:`, and `lcm:`, which are defined only on integers. It is also possible to evaluate a block an integral number of times. The protocol for integers appears in Figure 10.13.

`div: anInteger`	Answer the quotient of the receiver and the argument, with division rounding towards $-\infty$.
`mod: anInteger`	Answer the modulus of the receiver and the argument, with division rounding towards $-\infty$.
`gcd: anInteger`	Answer the greatest common denominator of the receiver and the argument.
`lcm: anInteger`	Answer the least common multiple of the receiver and the argument.
`timesRepeat: aBlock`	If the receiver is the integer n, send `value` to `aBlock` n times.

Figure 10.13: Protocol for integers

Fractions A `Fraction` represents a rational number as a fraction (numerator and denominator) in reduced form, meaning: (1) the numerator and denominator have no common factors; (2) if the numerator is zero, the denominator is one; and (3) the denominator is positive. Class `Fraction` responds to the message `num:den:`, creating a fraction with the given numerator and denominator.

Here is a session using `Fractions`. The call `(sqrtWithin: f1 eps)` asks for the square root of 2, to within 0.1. The result is the fraction $\frac{17}{12}$, or $1.41\bar{6}$; the square root of 2 is actually 1.4142+.

426a ⟨*transcript* 399⟩+≡ ◁422 426b▷

```
-> (val f1 (num:den: Fraction 2 1))
-> (val eps (num:den: Fraction 1 10))
1/10
-> (val f2 (sqrtWithin: f1 eps))
17/12
```

One can get better precision by decreasing `eps`. And of course it is more convenient and idiomatic to get fractions by dividing integers and by using `asFraction` than by sending messages to class `Fraction`.

426b ⟨*transcript* 399⟩+≡ ◁426a 427▷

```
-> (val eps (/ 1 100))
1/100
-> (val f2 (sqrtWithin: (asFraction 2) eps))
577/408
```

`asFraction`, in class `Float` 466b; in class `Fraction` 464b; in class `Integer` 461a; in class `Number` 459b; `sqrtWithin:` 460b

The implementation computes the square root of n by using the Newton-Raphson technique for finding a zero of the function $x^2 - n$. Newton-Raphson produces accurate results quickly; the approximation $\frac{577}{408}$ is accurate to five decimal places. Unfortunately, because μSmalltalk lacks arbitrary-precision integer arithmetic, the `Fraction` class is not much use for more ambitious computations; the denominators overflow too quickly. (Exercises 31 and 32 ask you to implement "large integers," which overflow only if your machine runs out of memory.)

Floating-point numbers A number in floating-point form is represented by a mantissa and exponent (of 10). That is, the `Float` having mantissa m and exponent e represents $m \times 10^e$; m and e can be negative. The mantissa has about 15 bits of precision; this restriction ensures we can multiply two mantissas without having the results overflow.

`Float` can be used as follows:

427 ⟨*transcript* 399⟩+≡ ◁426b

```
-> (val eps (asFloat (/ 1 10)))
1x10^-1
-> (sqrtWithin: (asFloat 2) eps)
14166x10^-4
```

The result is 1.4166, which is reasonably close to the correct answer of 1.4142+.

10.4 Extended example—Discrete-event simulation

Having been introduced to the μSmalltalk language and having seen the initial basis, we're ready to tackle a more ambitious example. The example in this section is big enough that you can see the interplay among classes and methods that characterizes object-oriented programs. In this example we look at a problem faced by our distinguished colleague Professor S.

Professor S's students are training robots to help urban search-and-rescue teams. For example, if firefighers cannot safely search a burning building, they might send in one of Professor S's robots. Unfortunately, fireproof robots are madly expensive, so Professor S's lab has has only two robots, and his students have to take turns. To make sure every student gets a turn, Professor S wants to limit each student to at most t minutes on any given robot; after t minutes, another student gets a turn. What should t be? More specifically, what value of t will minimize the time that the average student can expect to wait for a robot?

Professor S could experiment with different values of t in the robot lab, but the average waiting time is also affected by the number of students in the lab and by other conditions that are hard to reproduce, so it's not clear what the results would mean. And if some values of t are worse than others, the experiment is not fair to the students who are in the lab while those values are in force. The alternative we explore below is to write a program that *simulates* the lab—students arriving, waiting for robots, and using robots—and run the simulation multiple times with different values of t. Simulation has all sorts of advantages: it doesn't disrupt students; it's cheap enough to run many experiments; and the laboratory conditions are totally controlled and reproducible. But there's one huge caveat: we don't know if the simulation models what would really happen. In this section, we don't worry about realism; our goal is to learn to use Smalltalk.

asFloat,
in class Float 465g
in class Fraction 464b
in class Integer 461a
in class Number 459b
eps 426a
sqrtWithin: 460b

In the robot lab, the interesting events happen at discrete points in time: a student arrives and wants a robot; a student actually gets to use a robot; or because t minutes have elapsed, a student has to relinquish a robot. This situation calls for a *discrete-event simulation*. Discrete-event simulations can be used for many problems, including such problems as evaluating plans for handling baggage at a new airport, estimating traffic flow over a new highway, or deciding what inventory to keep in a new warehouse.

Other kinds of simulation work with continuous variables, like the voltage of electrons in a circuit, or the density of molecules in the atmosphere. These are *continuous-event simulations*, and the techniques used to implement them are very different from those we explore below.

10.4.1 Designing discrete-event simulations

Smalltalk's object-oriented style is a good fit for simulation. A full Smalltalk-80 system includes tools for modeling, viewing, and controlling simulations. Using these tools is so easy that even novice programmers can create interesting simulations. I've drawn on these tools to design a discrete-event simulation that highlights object-oriented programming techniques.

- If an entity in the system is allowed to take actions, like grabbing a robot, it is represented by a *simulation object*. In our example, each student is represented by a simulation object. A student takes such actions as asking for a robot or relinquishing a robot.
- If an entity represents a finite supply of some good or service—like a robot, a baggage cart, or a warehouse shelf—that entity is called a *resource*. The simulation classes that come with Smalltalk-80 provide special support that helps simulation objects acquire, release, or wait for resources. A resource might be represented by a single object, but it's also possible that a group of identical resources can be represented by a single object. An object representing a resource keeps track of the state of that resource as the simulation progresses. In our example, the only significant resource is the lab with its two fireproof robots.
- The overall simulation is orchestrated by an object, called "the simulation," whose class inherits from `Simulation`:
  - It keeps track of simulated time.
  - It schedules and runs every simulated event, always knowing what action is supposed to happen next.
  - It responds to requests for resources, and if a resource isn't available, it puts the requesting simulation object on a queue to wait.
  - It keeps track of whatever information about the simulation is important, so when the simulation is over, it can report conclusions. In our example, the simulation tracks the amount of time students spend waiting for robots.

As you walk through the design and implementation of the robot-lab simulation, keep an eye out for two salient aspects of the object-oriented style: you will see methods, like the `Simulation` instance methods, which are intended to be easy to reuse; and you will also see that, unlike in imperative programming, the actions needed to implement an algorithm tend to be "smeared out" over multiple methods of multiple classes, making the algorithm a bit difficult to follow.

Figure 10.14 sketches the protocol that I suggest for simulations. The protocol is adapted from similar protocols in the Smalltalk-80 blue book (Goldberg and Robson 1983):

- The first three methods of a `Simulation` instance make it possible to start, run, and end the simulation. A subclass typically adds extra initialization and finalization to the `startUp` and `finishUp` methods.
- The `enter:` and `exit:` methods allow a subclass to keep track of which "active" simulation objects are participating in the simulation.
- The `time-now` method and scheduling methods allow all participants to know the current time and to schedule future events.

Instance protocol for `Simulation`:

`startUp`	Initialize the simulation, including scheduling at least one event.
`proceed`	Simulate the next event.
`finishUp`	End the simulation and save (or print) the results.
`enter: anObject`	Notify the receiver that a new object (the argument) has entered the simulation.
`exit: anObject`	Notify the receiver that the argument has left the simulation.
`time-now`	Answer the current simulated time
`scheduleEvent:at: anEvent aTime`	Schedule the event `anEvent` to occur at the given simulated time. The `anEvent` object must respond to the `takeAction` message, which is sent to it when the scheduled time arrives.
`scheduleEvent:after: anEvent aTimeInterval`	Schedule the event to occur after the given (simulated) time interval will have passed.
`scheduleRecurringEvents:using: aClass aStream`	Get a time interval from `aStream` by sending it the `next` message, then schedule a new, anonymous event to occur after that interval. When the new event occurs, create a new simulation object by sending message `new` to `aClass`, then repeat indefinitely. The effect is a series of recurring events at time intervals given by `aStream`.
resource methods	(Every subclass of *simulation* provides subclass-specific methods that are used to acquire and release simulated resources.)

Global variable used by `Simulation`:

`ActiveSimulation`	Holds the value of the currently active `Simulation` object.

Figure 10.14: Partial instance protocol for class `Simulation`

- *Resource methods* are simulation-specific. They enable active objects to acquire and release resources, and they should be provided by a subclass of `Simulation`.
- Finally, the design assumes that only one simulation runs at a time. To provide easy accesss to that simulation, I define global variable `ActiveSimulation`.[9]

429 ⟨*simulation classes* 429⟩≡ 430▷

```
(val ActiveSimulation nil)
```

nil

Using this design, you can expect most of a simulation to be programmed with three kinds of messages:

- Messages from a simulation object to the simulation: notifying the simulation of entry and exit, requesting and releasing resources, and possibly scheduling an event or asking about the current time.

[9] In Smalltalk-80, `ActiveSimulation` would be a class variable (page 500), not a global variable.

- Messages from the simulation to a simulation object: granting access to a resource, or telling the simulation object to act. Granting access is simulation-specific, but to tell a simulation object to act, every simulation sends the `takeAction` message. This message is the only message to which all simulation objects must respond.
- Simulation-specific messages from either simulation or from simulation objects to resources or to other passive entities, telling them to change their states.

The rest of this section shows how to implement the `Simulation` class, how to implement a `RobotLabSimulation` subclass, and how to implement the simulation objects and resources that support the robot-lab simuilation.

10.4.2 Implementing the `Simulation` class

The methods for scheduling and simulating events are common to all simulations and should therefore be implemented just once, in the `Simulation` class. Several methods will be specialized by different subclasses, and simulation-specific resource methods are implemented only in subclasses.

To implement the protocol in Figure 10.14, we need only two instance variables:

- Variable `now` holds the current simulated time. A simulation is free to use any representation of time that answers the `Magnitude` protocol in Figure 10.12 on page 424. (A simulation needs only to know which of two times is smaller, because the event with the smallest time is the one that occurs the soonest.)
- Variable `eventQueue` holds events that have not yet taken place, but are scheduled to occur in the simulated future. The event queue may also hold events that are scheduled to occur at time `now`.

430 ⟨*simulation classes* 429⟩+≡ ◁429 433▷

```
(class Simulation Object
    (now eventQueue)
    (method time-now () now)
    ⟨more methods of class Simulation 431a⟩
)
```

The main invariant of a simulation is that at each point in time, the state of the objects in the simulation faithfully represents the state of the entities at the time stored in `now`. The states and the clock change only when there's an *event*. Events that are planned to occur in the simulated future are stored in `eventQueue`, which is a collection of events keyed by future time. The protocol for `eventQueue` is given in Figure 10.15, and its implementation is discussed further in Exercise 16 on page 510.

`Object` *B*

Instance protocol for `PriorityQueue`:

`isEmpty`	Answer `True` if and only if the receiver holds no events.
`at:put: aTime anEvent`	Add `anEvent` to the receiver, scheduling it to occur at time `aTime`.
`deleteMin`	Provided the receiver is not empty, answer an `Association` in which the `value` is an event that is contained in the receiver and has minimal time, and the `key` is the associated time.

Figure 10.15: Protocol for class `PriorityQueue`

Initializing, finalizing, and stepping a simulation

Initializing a simulation initializes the two instance variables and the global variable `ActiveSimulation`. To add initialization for its own private state, a subclass defines its own `startUp` method, which should send `(startUp super)`.

431a ⟨*more methods of class* `Simulation` 431a⟩≡ (430) 431b▷

```
(method startUp ()
    (set now 0)
    (set eventQueue (new PriorityQueue))
    (ifFalse: (isNil ActiveSimulation)
        [(error: self #multiple-simulations-active-at-once)])
    (set ActiveSimulation self)
    self)
```

Finalizing the simulation resets `ActiveSimulation` to `nil`.

431b ⟨*more methods of class* `Simulation` 431a⟩+≡ (430) ◁431a 431c▷

```
(method finishUp ()
    (set ActiveSimulation nil)
    self)
```

The `proceed` method simulates the next event in the queue.

431c ⟨*more methods of class* `Simulation` 431a⟩+≡ (430) ◁431b 431d▷

```
(method proceed () (locals event)
   (set event (deleteMin eventQueue))
   (set now (key event))
   (takeAction (value event)))
```

(This implementation is too simple-minded: it always sends `deleteMin` to `eventQueue`, but the client object that sends `proceed` can't know if `deleteMin` is safe. The `Simulation` protocol should be enriched so that clients can call `proceed` safely, as described in Exercise 22 on page 513.)

We define a method `runUntil:`, which runs events from the queue in order of increasing time until there are no more events—or until a time limit is reached. This is the method we use to run robot-lab simulations.

431d ⟨*more methods of class* `Simulation` 431a⟩+≡ (430) ◁431c 432a▷

```
(method runUntil: (timelimit)
    (startUp self)
    (while [(& (not (isEmpty eventQueue)) (<= now timelimit))]
       [(proceed self)])
    (finishUp self)
    self)
```

Tracking entry and exit of simulation objects

In a general simulation, the `enter:` and `exit:` methods don't do anything. To know what needs to be done when a simulation object enters or exits the simulation, we need a simulation-specific method. Such a method would be defined on a subclass of `Simulation`, but because a subclass is not *required* to do anything on entry or exit, trivial implementations of `enter:` and `exit:` are provided here.

432a ⟨*more methods of class* `Simulation` 431a⟩+≡ (430) ◁431d 432b▷

```
(method enter: (anObject) nil)
(method exit:  (anObject) nil)
```

Scheduling events

The fundamental scheduling operation around which the others are built is to schedule an event at a given time. An example would be to tell the simulation, "schedule the lab to open at 3:00PM." We schedule an event by using the `at:put:` method of class `PriorityQueue` to add the event to the event queue.

432b ⟨*more methods of class* `Simulation` 431a⟩+≡ (430) ◁432a 432c▷

```
(method scheduleEvent:at: (anEvent aTime)
    (at:put: eventQueue aTime anEvent))
```

It's often convenient to schedule an event not at an *absolute* time, but at a time that is *relative* to the current time. An example would be "schedule this student to relinquish her robot at time t minutes from now."

432c ⟨*more methods of class* `Simulation` 431a⟩+≡ (430) ◁432b 432d▷

```
(method scheduleEvent:after: (anEvent aTimeInterval)
    (scheduleEvent:at: self anEvent (+ now aTimeInterval)))
```

The most interesting scheduling method is one that schedules *recurring* events. This method takes two arguments:

at:put:, in class Catalan 821
in class Dictionary 453c
in class Keyed-Collection 452a
in class List 456d
nil 8
scheduleNext-Event 433

- An `eventFactory` provides the means of creating a new event, by sending it the message `new`. The `eventFactory` is typically (but not always) a class.
- A `timeStream` argument provides a sequence of intervals that should elapse between events. The next interval is obtained by sending the message `next` to a `timeStream`. In a full Smalltalk-80 system, times in a stream are computed using a random-number generator. For example, "random arrival times" are normally modeled using a random-number generator that uses a Poisson distribution.

To implement recurring events, we define a new class of simulation object which is called `RecurringEvents`. An object of class `RecurringEvents` is initialized with an `eventFactory` and a `timeStream`.

432d ⟨*more methods of class* `Simulation` 431a⟩+≡ (430) ◁432c

```
(method scheduleRecurringEvents:using: (eventFactory timeStream)
    (scheduleNextEvent (new:atNextTimeFrom: RecurringEvents eventFactory timeStream)))
```

An object of class `RecurringEvents` represents an infinite stream of future events. Every object in this class answers the `scheduleNextEvent` message, for which the protocol requires the receiver to remove the next event from itself and schedule it.

The implementation is a bit tricky. When the object receives `scheduleNextEvent`, it pulls the next time from the `timeStream`, but it schedules *itself* as a proxy for the real event that is supposed to occur at the next time. Then, when the scheduled event occurs, the proxy receives the `takeAction` message, and it responds by using the factory to create the real event that is supposed to occur at this time. In this way, we ensure the `new` message is sent to a factory object at the appropriate simulated time. Finally, `takeAction` finishes by scheduling the *next* recurring event. All this action is easier to code than to explain: the two methods together need only 5 lines of μSmalltalk.

433 ⟨*simulation classes* 429⟩+≡ ◁430 434a▷

```
(class RecurringEvents Object
    ; represents a stream of recurring events, each created from
    ; 'factory' and occurring at 'times'
    (factory times)
    (method scheduleNextEvent ()
        (scheduleEvent:after: ActiveSimulation self (next times)))
    (method takeAction ()
        (new factory)
        (scheduleNextEvent self))
    (class-method new:atNextTimeFrom: (eventFactory timeStream)
        (init:with: (new super) eventFactory timeStream))
    (method init:with: (f s) ; private
        (set factory f)
        (set times s)
        self)
)
```

The other methods (class method `new:atNextTimeFrom:` and instance method `init:with:`) implement the common pattern, first shown in class `FinancialHistory` in Figure 10.2, where we create an object by sending a message to a class method, which then uses a private instance method to initialize the new object.

ActiveSimulation 429
next, in class EveryNMinutes 442c
in class TwentyAtZero 441c
Object B
scheduleEvent:after: 432c

10.4.3 Implementing the robot-lab simulation

The implementation of a robot-lab simulation follows the plan sketched above:

- A single object of class `RobotLabSimulation` (a subclass of `Simulation`) orchestrates the simulation and keeps track of its state.
- Every simulation object that acts in the system is a student, each one of which is represented by an instance of class `Student`.
- The only resource we need to simulate is the lab itself, with its two robots. The lab is simulated by a single object of class `Lab`, and the `RobotLabSimulation` maintains a queue of students who are waiting to use the resource.

Because the `Lab` class is the simplest, we start there. We continue with `RobotLabSimulation`, and we finish with the most complex class, `Student`.

As you read the code, keep in mind the distinction between an event's being *scheduled* and that event's actually *occurring*. When an event is scheduled, it is simply added to the `eventQueue`; nothing else happens. Scheduling is the job of the `Simulation` superclass's scheduling methods. When an event *occurs* (when a simulation object receives `takeAction` or a factory object receives `new`), things happen, and the state of the simulation can change. Changing the state is the job of the `enter:` and `exit:` methods as well as the subclass-specific resource methods.

The class `Lab`

This class represents the state of the lab as a pair of Booleans, each of which says if a robot is available. Its protocol allows clients to check if there is a free robot (`hasARobot?`), get a robot (`takeARobot`), and give up a robot (`releaseRobot:`). All these methods are called when events occur, not when they are scheduled.

434a ⟨*simulation classes* 429⟩+≡ ◁433 434b▷

```
(class Lab Object
    (robot1free robot2free)
    (class-method new () (initLab (new super)))
    (method initLab () ; private
        (set robot1free true)
        (set robot2free true)
        self)
    (method hasARobot? () (| robot1free robot2free))
    (method takeARobot ()
         (if robot1free
              [(set robot1free false) 1]
              [(set robot2free false) 2]))
    (method releaseRobot: (t)
         (if (= t 1) [(set robot1free true)] [(set robot2free true)]))
)
```

The private `initLab` method ensures that in a new lab, both robots are available.

The class `RobotLabSimulation`

The class `RobotLabSimulation` maintains the state associated with a robot-lab simulation. A simulation carries a lot of internal state:

434b ⟨*simulation classes* 429⟩+≡ ◁434a 437a▷

```
(class RobotLabSimulation Simulation
    (time-limit          ; time limit for using one robot
     lab                 ; current state of the lab
     robot-queue         ; the line of students waiting for a robot
     students-entered    ; the number of students who have entered the lab
     students-exited     ; the number of students who have finished their work
                         ; and left the lab
     timeWaiting         ; total time spent waiting in line by students
                         ; who have finished
     student-factory     ; class used to create a new student when one enters
     interarrival-times  ; stream of times between student entries
    )
    ⟨methods of class RobotLabSimulation 435a⟩
)
```

if 446
Object B
Simulation 430

When introducing the robot-lab problem, we focused on the time limit t, which governs how long a student may use a robot while other students are waiting. But what happens in the lab is affected by more than just the time t. It also matters how many students there are, when students arrive at the lab, and how much time with a robot each student needs. All this information must be provided to the `RobotLabSimulation` object.

The number of students and the times at which they arrive are built into a single abstraction: a stream of *interarrival times*. (An interarrival time is the amount of time that elapses between the arrival of one student and the next.) The time needed by a student is built into a *factory* object that produces new students on demand. To create a simulation, then, we pass three parameters: a time limit t, a student factory `s`, and a stream of interarrival times `as`.

435a ⟨*methods of class* `RobotLabSimulation` 435a⟩≡ (434b) 435b▷

```
(class-method withLimit:student:arrivals: (t s as)
    (init-t:s:as: (new super) t s as))
(method init-t:s:as: (t s as) ; private method
    (set time-limit         t)
    (set student-factory    s)
    (set interarrival-times as)
    self)
```

The rest of the instance variables are initialized when the simulation is started by the `startUp` method. This method also initializes the superclass and schedules the (recurring) student arrivals.

435b ⟨*methods of class* `RobotLabSimulation` 435a⟩+≡ (434b) ◁435a 435c▷

```
(method startUp ()
    (set lab              (new Lab))
    (set students-entered 0)
    (set students-exited  0)
    (set timeWaiting      0)
    (set robot-queue      (new Queue))
    (startUp super)
    (scheduleRecurringEvents:using: self student-factory interarrival-times)
    self)
```

Finally, to prevent anybody from accidentally creating a simulation without initializing `time-limit`, `student-factory`, and `interarrival-times`, we redefine class method `new`:

435c ⟨*methods of class* `RobotLabSimulation` 435a⟩+≡ (434b) ◁435b 436a▷

```
(class-method new () (error: self #robot-lab-simulation-needs-arguments))
```

schedule-Recurring-Events:using: 432d
startUp 431a

Our `finishUp` method reports on the results of the simulation. We print just the information we care about: the number of students who have finished their work, the number left in line, and the total and average times spent waiting by the students who finished.

⟨*methods of class* RobotLabSimulation 435a⟩+≡ (434b) ◁435c 436b▷

```
(method finishUp ()
    (print   #Num-finished=)
    (print   students-exited)
    (printcomma self)
    (print   #left-waiting=)
    (print   (size robot-queue))
    (printcomma self)
    (print   #total-time-waiting=)
    (print   timeWaiting)
    (printcomma self)
    (print   #average-wait=)
    (println (div: timeWaiting students-exited))
    (finishUp super))
(method printcomma () ; private
    (print #,) (print space))
```

At entry and exit, the simulation updates its internal statistics:

⟨*methods of class* RobotLabSimulation 435a⟩+≡ (434b) ◁436a 436c▷

```
(method enter: (aStudent)
    (set students-entered (+ 1 students-entered)))
(method exit: (aStudent)
    (set students-exited  (+ 1 students-exited))
    (set timeWaiting      (+ timeWaiting (timeWaiting aStudent))))
```

The `enter:` and `exit:` methods are called when events occur, not when they are scheduled. The `exit:` method relies on the `Student` object to be able to tell us how much time it has spent waiting in the queue.

The robot-lab simulation defines two resource methods: the `requestRobotFor:` method requests a robot for a student, and the `releaseRobot:` method gives it up.

⟨*methods of class* RobotLabSimulation 435a⟩+≡ (434b) ◁436b 437b▷

```
(method requestRobotFor: (aStudent)
     (if (hasARobot? lab)
          [(beGrantedRobot: aStudent (takeARobot lab))]
          [(addLast: robot-queue aStudent)]))

(method releaseRobot: (aRobot)
    (releaseRobot: lab aRobot)
    (ifFalse: (isEmpty robot-queue)
       [(beGrantedRobot: (removeFirst robot-queue) (takeARobot lab))]))
```

These resource methods interact with a *queue*. If a student requests a robot when no robot is available, that student is put on the queue. And if, when a student releases a robot, there are other students waiting, the student who has been waiting the longest is removed from the queue and is granted use of the robot.

Instance protocol for `Student`:

`takeAction`	Simulate whatever action is appropriate to the receiver's current state.
`beGrantedRobot: aRobot`	Change the receiver's internal state to note that it now has a robot, and schedule a time at which to give up the robot.
`needsRobot?`	Answer whether the receiver still needs a robot.
`timeWaiting`	Answer the total amount of time the receiver has spent waiting for a robot.

Private methods for `Student`:

`timeNeeded`	This message is sent once, when an instance is created. The receiver answers the total amount of time it needs with a robot.
`relinquishRobot`	This method is sent when the receiver is using a robot, and time has arrived for the receiver to stop. In response, the receiver takes some action appropriate to its needs: if it is done with its work, it exits the simulation; otherwise it asks for more robot time.

Class protocol for `Student`:

`new`	The class creates a new `Student` whose status is `#awaiting-robot`, and the `Student` immediately enters the active simulation and requests a robot from it.

Figure 10.16: Protocol for `Student`

The robot queue is similar to the purely functional queue described in Section 3.6. But as is typical for Smalltalk, the queue is not a purely functional data structure; it is *mutable*. The operations we need from a queue (add at end and remove from beginning) are already provided by Smalltalk lists. But to help with debugging, we define a `Queue` subclass, which prints the list using the keyword `Queue`.

437a ⟨*simulation classes* 429⟩+≡ ◁434b 438▷

```
(class Queue List
    ()
    (method species     () Queue)
    (method printName   () (print #Queue))
)
```

Finally, the robot-simulation class exposes two public methods that make it possible for students to observe some of its state. The `time-limit` method makes it possible for a `Student` object to discover the time limit t, so it can relinquish its robot when the time limit expires. The `students-entered` method makes it easy to assign each `Student` object a unique number when it is created.

437b ⟨*methods of class* `RobotLabSimulation` 435a⟩+≡ (434b) ◁436c

```
(method time-limit       () time-limit)
(method students-entered () students-entered)
```

List 455
print,
in class Collection 450d
in class Float 466d
in class Fraction 464a
in class Large-Negative-Integer 819b
in class Large-Positive-Integer 819a
in class Natural 817b
in class Student 438

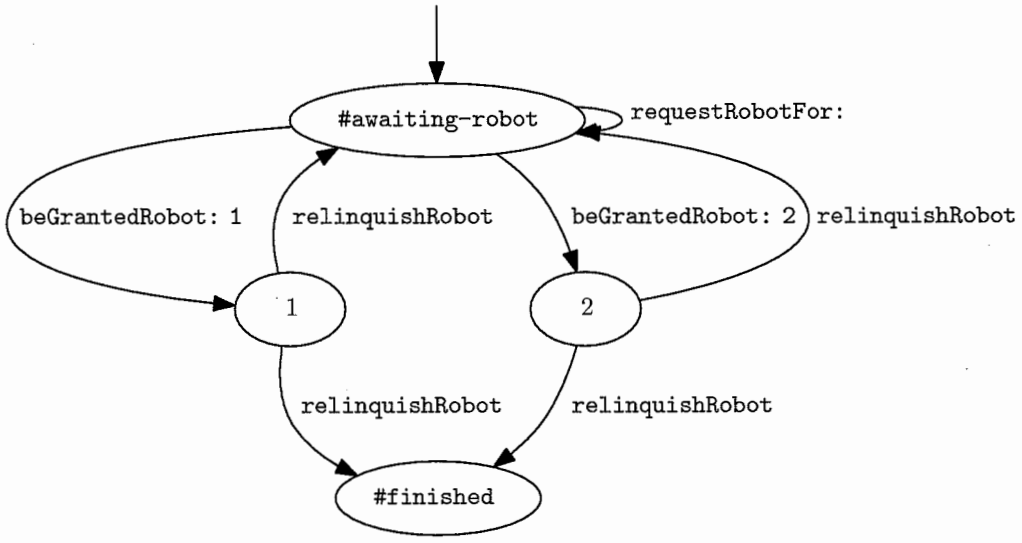


Figure 10.17: State-transition diagram for a Student

The class Student

In the robot-lab simulation, the active agents, also known as the simulation objects, are students. Each of these objects represents an individual who enters the lab, may wait in line, may use a robot, and so on. In the simulation, a student can be in one of four states: waiting for a robot, using robot 1, using robot 2, or finished. A diagram of these states, and of the messages that accompany transitions between them, is shown in Figure 10.17.

The Student class represents a student by six instance variables.

⟨*simulation classes* 429⟩+≡ ◁437a 441b▷

```
(class Student Object
    (number          ; uniquely identifies this student
     status          ; #awaiting-robot, #finished, or a robot number
     timeNeeded      ; total work time this student needs
     timeStillNeeded ; time remaining for this student
     entryTime       ; time at which this student enters the simulation
     exitTime        ; time at which this student exits the simulation
    )
    (method print () (print #<Student) (print space) (print number) (print #>))
    ⟨other methods of class Student 439a⟩
)
```

Object B
print,
  in class
    Collection
      450d
  in class Float
      466d
  in class Fraction
      464a
  in class Large-
    Negative-
    Integer
      819b
  in class Large-
    Positive-
    Integer
      819a
  in class Natural
      817b
space B

Here are some notes on the use of these instance variables:

- The `status` value indicates what the student is doing now, and also what it may do when it is next asked to do something via the `takeAction` method. The values correspond to the oval states in Figure 10.17.

Value	*State*
`#awaiting-robot`	Waiting for a robot (simulation will call `beGrantedRobot:`)
`1`	Using robot 1 (the next scheduled event is to release the robot)
`2`	Using robot 2 (the next scheduled event is to release the robot)
`#finished`	Finished (no more events will be scheduled for this student)

- Variable `timeNeeded` holds total amount of time the student needs with the robot in order to finish his or her lab work. Variable `timeStillNeeded` holds the amount of time left after whatever time the student has already spent with the robot. Our simulation assumes that having the robot time broken into chunks doesn't affect the amount of time needed. In practice this assumption is probably false.

- Variables `entryTime` and `exitTime` provide an easy way to compute the total time the student spent in the lab. The difference between the total time and `timeNeeded` is the time spent waiting, which is the data we're trying to gather. The data is provided to the simulation by the `timeWaiting` method.

439a ⟨*other methods of class* `Student` 439a⟩≡ (438) 439b▷

```
(method timeWaiting ()
    (- exitTime (+ entryTime timeNeeded)))
```

To create a `Student` object, we use the same pattern we saw in `FinancialHistory` and `DeductibleHistory`: a class method creates the instance, then executes a private method to initialize the object. Initialization is mostly straightforward: set the instance variables, enter the simulation, and ask for a robot. But there's a little something extra going on with `timeNeeded`:

439b ⟨*other methods of class* `Student` 439a⟩+≡ (438) ◁439a 440a▷

```
(method timeNeeded () (subclassResponsibility self))
(class-method new () (init (new super)))
(method init () ; private
  (set number          (+ 1 (students-entered ActiveSimulation)))
  (set status          #awaiting-robot)
  (set timeNeeded      (timeNeeded self))
  (set timeStillNeeded timeNeeded)
  (set entryTime       (time-now ActiveSimulation))
  (enter: ActiveSimulation self)
  (requestRobotFor: ActiveSimulation self)
  self)
```

ActiveSimulation 429
enter:, in class RobotLabSimulation 436b
in class Simulation 432a
init 441c
requestRobotFor: 436c

The value of the `timeNeeded` instance variable is obtained by sending the `timeNeeded` *message* to `self`. What's going on here? My design uses different subclasses of `Student` to represent students who have different needs for the robot. By delegating the knowledge of the need to a subclass, I make it easy to run simulations with students who have different needs.

After it requests a robot, a `Student` cannot do anything until the `RobotLabSimulation` tells it to. A `Student` is told to act by being sent the `takeAction` message. Its action depends on its `status`.

440a ⟨*other methods of class* `Student` 439a⟩+≡ (438) ◁439b 440b▷

```
(method takeAction ()
   (if (= status #awaiting-robot)
      [(requestRobotFor: ActiveSimulation self)]
      [(relinquishRobot self)]))
```

A student who needs a robot asks for one. A student who doesn't need a robot must already have one. That student should give up the robot, by sending himself the `relinquishRobot` message.

Relinquishing a robot always returns the robot to the active simulation, by sending the `releaseRobot:` message. The rest of the action depends on the student's needs.

- If he needs more time, he puts himself in the `#awaiting-robot` state, and he immediately requests the robot again. (He'll either wait in the queue, or in the special case where nobody else is waiting, he'll be granted the robot immediately. Because sending `requestRobotFor:` might result in an immediate message of `beGrantedRobot`, it's crucial that `status` be set to `#awaiting-robot` *before* `requestRobotFor:` is sent. Otherwise, the simulation might get into an inconsistent state in which the `Student` has been granted a robot but doesn't know it.)

- If the student has finished, he notes the current time as the `exitTime` from the simulation, and then he exits the simulation. Again, order of evaluation is crucial: sending `exit:` will result in the simulation sending `timeWaiting`, and if `exitTime` has not been set, a run-time error will occur.

ActiveSimulation 429
exit:, in class RobotLabSimulation 436b
in class Simulation 432a
if 446
releaseRobot:, in class Lab 434a
in class RobotLabSimulation 436c
requestRobotFor: 436c

These choices are shown graphically in Figure 10.17 by the two different arrows out of states 1 and 2, both labeled `relinquishRobot`.

⟨*other methods of class* `Student` 439a⟩+≡ (438) ◁440a 440c▷

```
(method relinquishRobot ()
     (releaseRobot: ActiveSimulation status)
     (if (needsRobot? self)
          [(set status #awaiting-robot)
           (requestRobotFor: ActiveSimulation self)]
          [(set status   #finished)
           (set exitTime  (time-now ActiveSimulation))
           (exit: ActiveSimulation self)]))
```

A student needs a robot if the time still needed is nonzero.

⟨*other methods of class* `Student` 439a⟩+≡ (438) ◁440b 441a▷

```
(method needsRobot? () (> timeStillNeeded 0))
```

The last remaining action in the `Student` class shows what happens when a student is granted use of a robot. He or she keeps the robot for as long as needed, or for the time limit t, whichever is smaller. The `beGrantedRobot:` method saves this time interval in the local variable `time-to-use`. The `Student` object then adjusts its internal `timeStillNeeded`, changes its status, and schedules itself on the event queue. When the scheduled event arrives, the student's `takeAction` method will relinquish the robot.

441a ⟨*other methods of class* `Student` 439a⟩+≡ (438) ◁440c

```
(method beGrantedRobot: (aRobot) (locals time-to-use)
     (set time-to-use (min: timeStillNeeded (time-limit ActiveSimulation)))
     (set timeStillNeeded (- timeStillNeeded time-to-use))
     (set status aRobot)
     (scheduleEvent:after: ActiveSimulation self time-to-use))
```

10.4.4 Running robot-lab simulations

To create a robot-lab simulation, we need a time limit, a student class, and a stream of interarrival times. We can then run the simulation for any given number of minutes. In a serious simulation, we would put a lot of effort into the classes that represent students' needs and arrival times. We would study how real students behave, create a probabilistic model, and code the model in Smalltalk. But studies are expensive, and force-feeding you a lot of probability and statistics would not help you learn about object-oriented techniques for implementing simulations. So I've chosen simplicity over realism; we'll make assumptions that oversimplify what happens in the real robot lab.

Our first simplifying assumption is that every student needs two hours of robot time, which we measure in minutes:

441b ⟨*simulation classes* 429⟩+≡ ◁438 441c▷

```
(class Student120 Student ; a student needing 120 minutes of robot time
    ()
    (method timeNeeded () 120)
)
```

Our second simplifying assumption is that we have 20 students, and they all pour into the lab the moment it opens (i.e., when the simulation starts). We need to embody this assumption as an infinite stream of interarrival times. In other words, we need an object which, when it is sent the `next` message, will answer `0`. But only 20 times! After responding 20 times with `0`, the object should respond to future `next` messages with a very large time—one large enough to exceed the duration of any reasonable simulation. The object will be an instance of class `TwentyAtZero`:

ActiveSimulation 429
if 446
init 439b
Object *B*
schedule-Event:after: 432c
Student 438

441c ⟨*simulation classes* 429⟩+≡ ◁441b 442b▷

```
(class TwentyAtZero Object ; Twenty arrivals at time zero
    (num-arrived)
    (class-method new () (init (new super)))
    (method init () (set num-arrived 0) self)
    (method next ()
         (if (= num-arrived 20)
             [99999]
             [(set num-arrived (+ 1 num-arrived))
              0]))
)
```

We use these classes, plus our implementation of `PriorityQueue` from Exercise 16, to create a simulation `sim30`. We then run the simulation for 20 simulated hours:

442a ⟨*simulation transcript* 442a⟩≡ 443▷

```
-> (use pqueue.smt) ; implementation of class PriorityQueue
-> (use sim.smt)    ; implementations of the simulation classes
-> (val sim30 (withLimit:student:arrivals: RobotLabSimulation 30 Student120
                                            (new TwentyAtZero)))
-> (runUntil: sim30 1200)
Num-finished=20, left-waiting=0, total-time-waiting=18900, average-wait=945
<RobotLabSimulation>
```

The robot lab was open long enough to serve all 20 students, so at the end, none were left waiting. But the time limit of 30 minutes does not seem to have worked out well; the average student waits for 945 minutes, spending nearly eight times as much time in line as working with a robot. The results of all four runs are as follows:

Time limit t	Students served	Students left waiting	Average wait time
30	20	0	945
60	20	0	810
90	20	0	945
120	20	0	540

If we want to minimize average waiting time, we do best to let each student monopolize a robot for a full two hours. This policy may not be fair, but it's efficient.

What if not all students are alike? Let's assume that only half the students need two hours each. The other half are accomplished roboticists and can finish their work in half an hour. Every time we create a new `Student`, we'll assume that the time needed by the new `Student` is 150 minutes minus the time needed by the previous student. That works out to `Student`s who alternate between needing 120 minutes and 30 minutes.

442b ⟨*simulation classes* 429⟩+≡ ◁441c 442c▷

```
(val last-student-needed 30) ; time needed by last created AlternatingStudent
(class AlternatingStudent Student
    ()
    (method timeNeeded ()
        (set last-student-needed (- 150 last-student-needed))
        last-student-needed)
)
```

Object 𝓑
RobotLab-Simulation 434b
runUntil: 431d
Student 438
Student120 441b
TwentyAtZero 441c

In Smalltalk-80 we would store `last-student-needed` in a *class variable*, which would be shared among all instances of `AlternatingStudent` (see Exercise 36).

Let's also assume that the students know that there are only two robots, so they don't all crowd into the lab when it opens. Instead, they arrive every 35 minutes. And to keep the implementation simple, we won't cap the number of students at 20; instead, we assume that as long as the lab is open, students keep coming.

An object of class `EveryNMinutes` always returns the same interarrival time `n`, which is passed as a parameter to class method `new:`.

442c ⟨*simulation classes* 429⟩+≡ ◁442b

```
(class EveryNMinutes Object
    (interval)
    (class-method new: (n) (init: (new super) n))
    (method init: (n) (set interval n) self)
    (method next () interval)
)
```

To make these new simulations easier to run, we create an auxiliary helper class `AlternatingLabSim`. It's a subclass of `RobotLabSimulation`, and it has an extra class method which knows to use `AlternatingStudent` every 35 minutes. Again, we run it four times:

443 ⟨*simulation transcript* 442a⟩+≡ ◁442a

```
-> (class AlternatingLabSim RobotLabSimulation
      ()
      (class-method runWithLimit: (n)
         (runUntil: (withLimit:student:arrivals: super n AlternatingStudent
                                                 (new: EveryNMinutes 35))
                    1200))
   )
-> (runWithLimit: AlternatingLabSim 30)
Num-finished=30, left-waiting=2, total-time-waiting=1095, average-wait=36
<AlternatingLabSim>
-> (runWithLimit: AlternatingLabSim 60)
Num-finished=30, left-waiting=2, total-time-waiting=1235, average-wait=41
<AlternatingLabSim>
-> (runWithLimit: AlternatingLabSim 90)
Num-finished=29, left-waiting=3, total-time-waiting=1190, average-wait=41
<AlternatingLabSim>
-> (runWithLimit: AlternatingLabSim 120)
Num-finished=30, left-waiting=2, total-time-waiting=1120, average-wait=37
<AlternatingLabSim>
```

The results for the four runs this time are:

Time limit t	Students served	Students left waiting	Average wait time
30	30	2	36
60	30	2	41
90	29	3	41
120	30	2	37

With these different students, there's no time limit t that is clearly superior, although both the "rapid turnover" and "hold for two hours" policies are about 12% better than intermediate limits. Because the simulation is so unrealistic, we shouldn't draw any conclusions.

Alternating-Student 442b
EveryNMinutes 442c
RobotLab-Simulation 434b
runUntil: 431d

10.4.5 Summary and analysis

Our simulation omits too many details. For example, a real student who enters the lab and finds a long line may *balk*, i.e., he may leave and try again later. We don't consider the cost of interruptions; a student whose work is broken into several sessions may need more time with the robots.[10] "Average time waiting" is not a definitive measure for comparing time limits, because it values everyone's time equally. But Professor S might prefer a policy under which students who need less time don't have to wait as long as students who need more time.

[10] It's also possible that students who are interrupted spend more time thinking, after which they may need to spend *less* time fiddling with robots.

Most importantly, our simulations make bogus assumptions about needs and about arrival times—and these assumptions probably have a decisive effect on the results. We might build into the simulation a list of needs and arrival times obtained by observing real students, or we might generate them randomly according to a distribution that we believe reflects the real situation.

Many of the problems enumerated above can be addressed by making modest changes to the simulation code. Suggestions for such changes appear in Exercise 18.

Although our simulation does not accurately model real students working in real labs, it *does* demonstrate a good way to organize an object-oriented simulation. To understand the organization deeply, you will need to do some exercises. But we can jump-start your understanding by looking at the organization through the lens of a single computation: the algorithm executed when a new student enters the lab. In a typical imperative language like C or Impcore, we might write a single "new student" procedure that does this:

- Allocate memory for the student and initialize its fields. Increment the number of students in the simulation. Finally check to see if a robot is available. If a robot is available, assign it to the student and add a "robot time expires" event to the event queue. If no robot is available, put the student on the queue for the robot.

Let's contrast this single "new student" procedure with the way the same computation is done in the Smalltalk code:

1. An object of class `RecurringEvents` sends a `new` message to its local factory, which is the class object `Student120`.

2. The `new` message is dispatched to class `Student`, which calls `new super`, which is dispatched to `Object`. Space is allocated for the object and its instance variables. The `new` method in class `Student` then sends `init` to the new object.

3. The `init` method on class `Student` initializes the instance variables, which includes sending `timeNeeded` to `self`, which dispatches on the `Student120` class, answering 120. The init method then sends `enter:` to the active simulation.

4. The `enter:` method on class `RobotLabSimulation` increments the number of students in the simulation.

5. The `init` method on class `Student` finishes by sending `requestRobotFor:` to the active simulation.

6. The `requestRobotFor:` method on class `RobotLabSimulation` checks to see if a robot is available. If a robot is available, it notes that the robot is no longer free, then sends `beGrantedRobot:` to the student; otherwise it adds the student to the robot queue.

7. The `beGrantedRobot:` method on class `Student` notes that the student is using the robot, calculates a `time-to-use`, then sends `scheduleEvent:after:` to the active simulation.

8. The `scheduleEvent:after:` method dispatches to the superclass `Simulation`, which in turn dispatches to `scheduleEvent:at:`, which finally puts the "robot time expires" event on the event queue.

This example illustrates what's hard about object-oriented programming: the algorithm, which the imperative programmer thinks of as one simple sequence of actions, ends up being "smeared out" over nine methods defined on four classes.

10.5 Implementations of μSmalltalk's predefined classes and objects

This section presents the implementations of the classes and objects that are predefined in μSmalltalk. Almost all of these classes are defined in μSmalltalk itself; only three classes are built into the interpreter:

- `Object`, because it has no superclass, and all later objects inherit from it
- `UndefinedObject`, because the interpreter needs `nil` internally
- `Class`, because all metaclasses must inherit from it

The remaining predefined classes are implemented in ordinary μSmalltalk, although some use a lot of primitive methods. This section presents the implementations.

10.5.1 Classes and objects built into the interpreter

Among the built-in classes and objects, only `Object` and `UndefinedObject` warrant discussion here. Most of the methods of `Object`—printing, class membership, equality testing, and so on—are primitive, but `isNil` and `notNil` can be defined in ordinary μSmalltalk. The way the messages `isNil` and `notNil` are defined is an object lesson in the use of method dispatch to implement case analysis.

Case analysis should use method dispatch, not conditionals

The `isNil` method could be implemented like this:

```
(method isNil () (= self nil))    ; embarassing code
```

A real Smalltalk programmer sneers at this code. The proper way to implement case analysis is not by using a conditional test or the `ifTrue:ifFalse:` message. The proper technique is to use method dispatch.

For `isNil` there are only two possible cases: the method should answer `true` or `false`. We arrange that on the `nil` object, the `isNil` method answers `true`, and that on all other objects, it answers `false`. We achieve this arrangement by only two method definitions: one on class `Object`, which all other classes inherit, and one on class `UndefinedObject`, which is used only to answer messages send to `nil`. We implement `notNil` the same way.

Each method is defined twice, once in `Object` and once in `UndefinedObject`. The definitions in class `Object` are equivalent to

```
(method isNil  () false)
(method notNil () true)
```

In `UndefinedObject`, they are:

```
(method isNil  () true)
(method notNil () false)
```

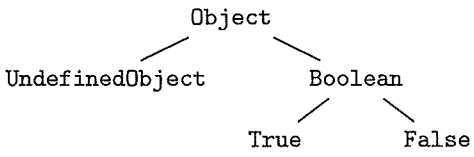
We assume that no other class in the hierarchy will redefine these messages, and that no subclasses will be added to `UndefinedObject`.

When the `isNil` message is sent to `nil`, the method search begins in the class of `nil`, which is `UndefinedObject`; it immediately succeeds and returns `true`. Similarly, `notNil`, when sent to `nil`, returns `false`. Thus, both messages respond correctly when sent to `nil`. On the other hand, when `isNil` is sent to *any other* object, the method search starts in the class of that object and goes all the way up to class `Object`, where it finally succeeds; the `isNil` method in `Object`, of course, returns `false`. Similarly, the `notNil` message, sent to any value other than `nil`, returns `true`.

If you take object-oriented programming seriously, you will *never use an explicit conditional if you can achieve the same effect using method dispatch.* Not only is it better style to use method dispatch, but in many implementations, method dispatch is much more efficient than conditional tests.

10.5.2 Booleans and blocks

Like Smalltalk-80, μSmalltalk has no built-in control structures except message passing. Conditional execution is implemented by methods on the `Boolean` class, together with subclasses `True` and `False`.



Booleans

The class `Boolean` is abstract. The `True` and `False` subclasses each define `ifTrue:ifFalse:`, and the `Boolean` class implements everything else in terms of `ifTrue:ifFalse`.

false 𝓑
ifTrue:ifFalse:,
in class False 447c
in class False 447a
in class True 447b
not,
in class False 447c
in class True 447b
Object 𝓑
true 𝓑

⟨*additions to the μSmalltalk initial basis* 446⟩≡ 447a▷

```
(class Boolean Object ()
    (method ifTrue:ifFalse: (trueBlock falseBlock) (subclassResponsibility self))

    (method ifFalse:ifTrue: (falseBlock trueBlock)
        (ifTrue:ifFalse: self trueBlock falseBlock))
    (method ifTrue:  (trueBlock)  (ifTrue:ifFalse: self trueBlock []))
    (method ifFalse: (falseBlock) (ifTrue:ifFalse: self [] falseBlock))

    (method not ()          (ifTrue:ifFalse: self [false]          [true]))
    (method eqv: (aBoolean) (ifTrue:ifFalse: self [aBoolean]       [(not aBoolean)]))
    (method xor: (aBoolean) (ifTrue:ifFalse: self [(not aBoolean)] [aBoolean]))

    (method & (aBoolean) (ifTrue:ifFalse: self [aBoolean] [self]))
    (method | (aBoolean) (ifTrue:ifFalse: self [self] [aBoolean]))

    (method and: (alternativeBlock) (ifTrue:ifFalse: self alternativeBlock [self]))
    (method or:  (alternativeBlock) (ifTrue:ifFalse: self [self] alternativeBlock))

    (method if (trueBlock falseBlock) (ifTrue:ifFalse: self trueBlock falseBlock))
)
```

Like `isNil` and `notNil`, `ifTrue:ifFalse:` does case analysis using method dispatch. Classes `True` and `False` each have a single object; these objects are named `true` and `false`, respectively. Boolean-valued methods return one of these two objects. Thus, the `ifTrue:ifFalse:` message is always sent to one of these objects. To implement conditional control, we simply define its method differently in each subclass.

447a ⟨*additions to the μSmalltalk initial basis* 446⟩+≡ ◁446 448a▷

```
(class True Boolean ()
  (method ifTrue:ifFalse: (trueBlock falseBlock) (value trueBlock))
)
(class False Boolean ()
  (method ifTrue:ifFalse: (trueBlock falseBlock) (value falseBlock))
)
```

So, in the expression

```
(ifTrue:ifFalse:
   (includesKey: incomes reason)
   [(at: incomes source)]
   [0])
```

the second line evaluates to either `true` or `false`. If `true`, the search for method `ifTrue:ifFalse:` begins in class `True` and immediately succeeds, sending `value` to the block `[(at: incomes source)]`; if `false`, it begins in class `False`, and again succeeds, sending `value` to `[0]`.

In real Smalltalk systems, to improve efficiency, the `True` and `False` classes override more of the `Boolean` methods.

447b ⟨*potential additions to the μSmalltalk initial basis* 447b⟩≡ 447c▷

```
(class True Boolean ()
  (method ifTrue:ifFalse: (trueBlock falseValue) (value trueBlock))
  (method ifTrue:  (trueBlock) (value trueBlock))
  (method ifFalse: (falseBlock) nil)

  (method & (aBoolean) aBoolean)
  (method | (aBoolean) self)
  (method not () false)
  (method eqv: (aBoolean) aBoolean)
  (method xor: (aBoolean) (not aBoolean))
)
```

447c ⟨*potential additions to the μSmalltalk initial basis* 447b⟩+≡ ◁447b

```
(class False Boolean ()
  (method ifTrue:ifFalse: (trueBlock falseBlock) (value falseBlock))
  (method ifTrue:  (trueBlock) nil)
  (method ifFalse: (falseBlock) (value falseBlock))

  (method & (aBoolean) self)
  (method | (aBoolean) aBoolean)
  (method not () true)
  (method xor: (aBoolean) aBoolean)
  (method eqv: (aBoolean) (not aBoolean))
)
```


Boolean	446
false	*B*
nil	*B*
not	446
true	*B*
value	452b

Blocks

The `value` method is primitive, but the `while`, `whileTrue:`, and `whileFalse:` methods are easily defined in ordinary μSmalltalk.

448a ⟨*additions to the μSmalltalk initial basis* 446⟩+≡ ◁447a 482a▷

```
(class Block Object
    () ; internal representation
    (method value primitive value)
    (method whileTrue: (body)
        (ifTrue:ifFalse: (value self)
            [(value body)
             (whileTrue: self body)]
            [nil]))
    (method while (body) (whileTrue: self body))
    (method whileFalse: (body)
         (ifTrue:ifFalse: (value self)
             [nil]
             [(value body)
              (whileFalse: self body)]))
)
```

10.5.3 The `Collection` hierarchy

This section presents the implementation of the `Collection` hierarchy, which reuses methods aggressively. Since `Collection` is an abstract class, it's not useful to send `new` to it, so why have it? Does it serve only to define an interface? No. `Collection` defines methods that are inherited by subclasses. In fact, `Collection` defines implementations of all but three of the messages in Figure 10.7; you can add a new collection to the hierarchy by defining `do:`, `add:`, `remove:ifAbsent:`, and `species`. Once these methods are defined, the default implementations of the remaining methods, shown below, become useful.

⟨*collection classes* 448b⟩≡ (495d) 451▷

```
(class Collection Object
  () ; abstract
  (method do:     (aBlock)    (subclassResponsibility self))
  (method add:    (newObject) (subclassResponsibility self))
  (method remove:ifAbsent: (oldObject exnBlock)
                              (subclassResponsibility self))
  (method species ()          (subclassResponsibility self))
  ⟨other methods of class Collection 448c⟩
)
```

To create a singleton collection, we use `new` and `add:`.

⟨*other methods of class* `Collection` 448c⟩≡ (448b) 449a▷

```
(class-method with: (anObject) (locals temp)
    (set temp (new self))
    (add: temp anObject)
    temp)
```

Given `add:`, `remove:ifAbsent:`, and `do:`, we can easily implement `remove:`, `addAll:`, and `removeAll:`. When any of these messages is sent to an object of a subclass, the message dispatches to one of the methods defined here. When these methods in turn send the messages `do:`, `remove:ifAbsent:`, and `add:`, they dispatch to the implementations of `do:`, `remove:ifAbsent:`, and `add:`, that are defined in the subclass.

449a ⟨*other methods of class* `Collection` 448c⟩+≡ (448b) ◁448c 449b▷

```
(method remove: (oldObject)
    (remove:ifAbsent: self oldObject [(error: self #tried-to-remove-absent-object)]))
(method addAll: (aCollection)
    (do: aCollection (block (x) (add: self x)))
    aCollection)
(method removeAll: (aCollection)
    (do: aCollection (block (x) (remove: self x)))
    aCollection)
```

In addition to these mutators, the `Collection` protocol in Figure 10.7 on page 418 defines a host of observers, including `isEmpty` and `size`, among others. The default implementations given here search through all the elements of the collection using `do:`. Using a linear search to compute `size`, for example, may seem terribly inefficient, but if a subclass knows a more efficient way to compute the number of elements, it should redefine the `size` method. And for some collections, like `List`, there really is no better way to compute the size than to visit all the elements. Similar remarks apply to the other methods.

449b ⟨*other methods of class* `Collection` 448c⟩+≡ (448b) ◁449a 450a▷

```
(method isEmpty () (= (size self) 0))
(method size () (locals temp)
    (set temp 0)
    (do: self (block (_) (set temp (+ temp 1))))
    temp)
(method occurrencesOf: (anObject) (locals temp)
    (set temp 0)
    (do: self (block (x)
       (ifTrue: (= x anObject) [(set temp (+ temp 1))])))
    temp)
(method includes: (anObject) (< 0 (occurrencesOf: self anObject)))
(method detect: (aBlock)
    (detect:ifNone: self aBlock [(error: self #no-object-detected)]))
(method detect:ifNone: (aBlock exnBlock) (locals answer searching)
    (set searching true)
    (do: self (block (x)
        (ifTrue: (and: searching [(value aBlock x)])
             [(set searching false)
              (set answer x)])))
    (if searching exnBlock [answer]))
```

The implementation of `detect:ifNone:` is genuinely inefficient: even if the `answer` is found relatively early in the search, there is no way to stop the `do:` method from iterating through all the elements. Full Smalltalk-80 addresses this problem with a control construct that I have left out of μSmalltalk. This construct, called "nonlocal return," can terminate a method like `detect:ifNone:` without waiting for `do:` to finish.

In addition to mutators and observers, the `Collection` protocol also provides iterators. These iterators have a lot in common with some of μScheme's higher-order functions on lists. For example, `inject:into:` is very similar to μScheme's `foldl`. But here is another case where Smalltalk's inheritance offers an advantage: unlike `foldl`, `inject:into:` works on *any* collection, not just on lists.

⟨*other methods of class* `Collection` 448c⟩+≡ (448b) ◁449b 450b▷

```
(method inject:into: (thisValue binaryBlock)
   (do: self (block (x) (set thisValue (value binaryBlock x thisValue))))
   thisValue)
```

The methods `select:`, `reject:` and `collect:` are related to μScheme's `filter` and `map` functions. Again, they work on all collections, not just on lists. The implementations show how we use `species`, which fulfills the protocol's requirement to create "a new collection like the receiver."

⟨*other methods of class* `Collection` 448c⟩+≡ (448b) ◁450a 450c▷

```
(method select: (aBlock) (locals temp)
   (set temp (new (species self)))
   (do: self (block (x) (ifTrue: (value aBlock x) [(add: temp x)])))
   temp)
(method reject: (aBlock)
   (select: self (block (x) (not (value aBlock x)))))
(method collect: (aBlock) (locals temp)
   (set temp (new (species self)))
   (do: self (block (x) (add: temp (value aBlock x))))
   temp)
```

To get a set that contains all of a collection's elements, we iterate.

⟨*other methods of class* `Collection` 448c⟩+≡ (448b) ◁450b 450d▷

```
(method asSet () (locals temp)
     (set temp (new Set))
     (do: self (block (x) (add: temp x)))
     temp)
```

Finally, `Collection` defines its own `print` method. By default, a collection prints as the name of its class, followed by a parenthesized list of its elements.

⟨*other methods of class* `Collection` 448c⟩+≡ (448b) ◁450c

```
(method print ()
    (printName self)
    (print lparen)
    (do: self (block (x) (print space) (print x)))
    (print space)
    (print rparen)
    self)
(method printName () (print #Collection))
```

Subclasses are encouraged to override `printName`.

Set

Set is a concrete class: it has instances. Like most other Smalltalk collections, Set is mutable; for example, sending add: to a set changes the set. A Set object is represented by a List, contained in its instance variable members. This structure makes Set a *client* of List, not a subclass or superclass.

Set and List share the Collection protocol, and two of the Set methods send the corresponding message to members, a technique called *delegation*.

451 ⟨*collection classes* 448b⟩+≡ (495d) ◁448b 452a▷

```
(class Set Collection
    (members)  ; list of elements
    (class-method new () (initSet (new super)))
    (method initSet   () (set members (new List)) self) ; private
    (method do: (aBlock) (do: members aBlock))
    (method remove:ifAbsent: (item exnBlock)
        (remove:ifAbsent: members item exnBlock))
    (method add: (item)
        (ifFalse: (includes: members item) [(add: members item)])
        item)
    (method species   () Set)
    (method printName () (print #Set))
    (method asSet     () self)
)
```

Set is our first concrete collection. To see better how collections are implemented, let's look at each method.

- The class method new initializes the representation (to the empty list) by means of private instance method initSet.
- Two of the four required methods, do: and remove:ifAbsent:, are delegated to List.
- The required add: method cannot be delegated to List, because a set must avoid duplicates in members. The add: method avoids duplicates by sending the includes: message to members, adding item to members only if it is not already present. The method add: could have sent the includes: message to self; but because List might have an includes: method that is more efficient than the default, Set sends includes: to members instead.
- The final required method, species, simply answers Set.
- After defining initialization methods and the required methods, Set also redefines printName as recommended, and for efficiency, it redefines asSet.

In addition to the methods shown in the class definition, Set inherits size, isEmpty, includes:, print, and other methods from Collection.

KeyedCollection and Association

A keyed collection provides access to key-value pairs. Any subclass must define methods `associationsDo:`, which replaces `do:`, and `at:put:`, which sometimes replaces `add:`.

452a ⟨*collection classes* 448b⟩+≡ (495d) ◁451 452b▷

```
(class KeyedCollection Collection
    ()  ; abstract class
    (method at:put: (key value)       (subclassResponsibility self))
    (method associationsDo: (aBlock)  (subclassResponsibility self))
    ⟨other methods of class KeyedCollection 452c⟩
)
```

Method `associationsDo:` visits all the key-value pairs in a keyed collection. A key-value pair is represented by an object of class `Association`.

452b ⟨*collection classes* 448b⟩+≡ (495d) ◁452a 453b▷

```
(class Association Object
   (key value)
   (class-method key:value: (a b) (setKey:value: (new self) a b))
   (method setKey:value: (x y) (set key x) (set value y) self) ; private
   (method key    ()  key)
   (method value  ()  value)
   (method key:   (x) (set key   x))
   (method value: (y) (set value y))
)
```

`Associations` are mutable.

Given `associationsDo:`, we implement the required `do:` method by iterating over associations.

452c ⟨*other methods of class* KeyedCollection 452c⟩≡ (452a) 452d▷

```
(method do: (aBlock)
    (associationsDo: self (block (x) (value aBlock (value x)))))
```


Every method in the "at" family, as well as `includesKey:`, is ultimately implemented on top of `associationsAt:ifAbsent:`, which uses `associationsDo:`.

⟨*other methods of class* KeyedCollection 452c⟩+≡ (452a) ◁452c 453a▷

```
(method at: (key)
    (at:ifAbsent: self key [(error: self #key-not-found)]))
(method at:ifAbsent: (key exnBlock)
    (value (associationAt:ifAbsent: self key
               [(key:value: Association nil (value exnBlock))])))
(method includesKey: (key)
    (isKindOf: (associationAt:ifAbsent: self key []) Association))
(method associationAt: (key)
    (associationAt:ifAbsent: self key [(error: self #key-not-found)]))
(method associationAt:ifAbsent: (key exnBlock) (locals finishBlock)
    (set finishBlock exnBlock)
    (associationsDo: self (block (x)
        (ifTrue: (= (key x) key) [(set finishBlock [x])])))
    (value finishBlock))
```

In implementing `associationAt:ifAbsent:`, we again use the power of method dispatch to avoid conditional tests; `finishBlock` is set either to the exception block or to a block that returns the pair sought. We use the same technique for finding keys.

453a ⟨*other methods of class* `KeyedCollection` 452c⟩+≡ (452a) ◁452d

```
(method keyAtValue: (value)
    (keyAtValue:ifAbsent: self value [(error: self #value-not-found)]))
(method keyAtValue:ifAbsent: (value aBlock) (locals finishBlock)
    (set finishBlock aBlock)
    (associationsDo: self (block (x)
        (ifTrue: (= (value x) value) [(set finishBlock [(key x)])])))
    (value finishBlock))
```

Subclasses can avoid linear search by redefining methods `associationAt:ifAbsent:` and `keyAtValue:ifAbsent:`.

`Dictionary`

A `Dictionary` is the simplest and least specialized of the keyed collections. If all μSmalltalk objects could be hashed, we would want to use a hash table. Instead we use a simple list of `Associations`.

453b ⟨*collection classes* 448b⟩+≡ (495d) ◁452b 454▷

```
(class Dictionary KeyedCollection
    (table) ; list of Associations
    (class-method new ()       (initDictionary (new super)))
    (method initDictionary () (set table (new List)) self) ; private
    (method printName ()       (print #Dictionary))
    (method species ()         Dictionary)
    ⟨other methods of class Dictionary 453c⟩
)
```

Iteration over associations can be delegated to the list of pairs. To implement `at:put:`, we search for the association containing the given key. If we find such an association, we mutate its value. If we find no such association, we add one.

453c ⟨*other methods of class* `Dictionary` 453c⟩≡ (453b) 453d▷

```
(method associationsDo: (aBlock) (do: table aBlock))
(method at:put: (key value) (locals tempassoc)
    (set tempassoc (associationAt:ifAbsent: self key []))
    (if (isNil tempassoc)
        [(add: table (key:value: Association key value))]
        [(value: tempassoc value)])
    value)
```

Because a dictionary requires not just a value but also a key, we can't give a sensible semantics to `add:`.

453d ⟨*other methods of class* `Dictionary` 453c⟩+≡ (453b) ◁453c

```
(method add: (_) (error: self #can't-just-add:-to-a-Dictionary))
```

SequenceableCollection

The abstract class SequenceableCollection defines methods used by KeyedCollections whose keys are consecutive integers. We redefine at:ifAbsent: to exploit the fact that keys are consecutive integers; the new version makes it possible for some subclasses, notably arrays, to avoid building Associations.

⟨*collection classes* 448b⟩+≡ (495d) ◁453b 455▷

```
(class SequenceableCollection KeyedCollection
    () ; abstract class
    (method firstKey () (subclassResponsibility self))
    (method lastKey  () (subclassResponsibility self))
    (method last     () (at: self (lastKey  self)))
    (method first    () (at: self (firstKey self)))
    (method at:ifAbsent: (index exnBlock) (locals current resultBlock)
        (set resultBlock exnBlock)
        (set current (firstKey self))
        (do: self (block (v)
            (ifTrue: (= current index) [(set resultBlock [v])])
            (set current (+ current 1))))
        (value resultBlock))
)
```

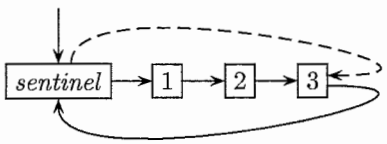

List

As in μScheme, we represent a list using cons cells. But abstractly, a μSmalltalk list is different from a μScheme list:

- In μSmalltalk, a list is a *mutable* type.
- In μSmalltalk, we can grow and shrink a list at the *end* as well as at the beginning.

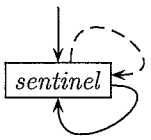
To support efficient insertion and deletion at either end of a list, we represent the list using a *circular* list of cons cells. A special cons cell marks the beginning and end of the list; it is called a *sentinel*. Using a sentinel is a standard technique that often simplifies the implementation of a data structure (Sedgewick 1988). Because we are programming in μSmalltalk, we can attach all the special-case end-of-list code to the sentinel, as methods of class ListSentinel. As a result, we implement lists without even one conditional that checks for end of list.

Just as in μScheme, a cons cell holds two values: a car and a cdr. Unlike in μScheme, the cdr *always* points to another cons cell. This invariant holds because every list is circularly linked. For example, here is a list containing the elements 1, 2, and 3:



The sentinel contains *two* pointers: the usual cdr, which it inherits from class Cons, points to the remaining (non-sentinel) elements of the list; the pred field (shown as a dashed line), which is private to class ListSentinel, points to the sentinel's *predecessor*.

A sentinel's predecessor is normally the last element of its list, but if a list is empty then both fields of the sentinel point to the sentinel itself:



Each cons cell, including the sentinel, responds to the following protocol:

`car`	Answer the car of the receiver.
`cdr`	Answer the cdr of the receiver.
`car: anObject`	Set the receiver's car and answer the receiver.
`cdr: anObject`	Set the receiver's cdr and answer the receiver.
`pred: aCons`	Notify the receiver that its predecessor is the cons cell `aCons`.
`deleteAfter`	Delete the cons cell that the receiver's cdr points to. Answer the car of that cons cell.
`insertAfter: anObject`	Insert a new cons cell after the receiver, letting the new cons cell's car point to `anObject`. Answer `anObject`.
`do: aBlock`	For each cons cell from the receiver to the sentinel, send the car of that cons cell to `aBlock`.
`rejectOne:ifAbsent:withPred: aBlock exnBlock aCons`	Starting at the receiver, search down the list for a cons cell satisfying `aBlock`. That is, we search for a cons cell `c` such that `(value aBlock c)` is true. If we find such a cell, we remove it from the list. If we don't find such a cell, we answer `(value exnBlock)`. As a precondition, the argument `aCons` *must* be the predecessor of the receiver.

In addition, a sentinel (and only a sentinel) can respond to this message:

`pred`	Answer the predecessor of the receiver.

A list is represented by a pointer to its sentinel.

455 ⟨*collection classes* 448b⟩+≡ (495d) ◁454 458b▷

```
⟨classes that define cons cells and sentinels 456e⟩
(class List SequenceableCollection
    (sentinel)
    (class-method new ()        (sentinel: (new super) (new ListSentinel)))
    (method sentinel: (s)       (set sentinel s) self) ; private
    (method isEmpty   ()        (= sentinel (cdr sentinel)))
    (method last      ()        (car (pred sentinel)))
    (method do:       (aBlock)  (do: (cdr sentinel) aBlock))
    (method species   ()        List)
    (method printName ()        (print #List))
    ⟨other methods of class List 456a⟩
)
```

The method `addLast:` mutates a list by adding an element to the end. This means inserting an element just after the predecessor of the sentinel. Similarly, `addFirst:` inserts just after the sentinel itself. Having a sentinel means there is no special-case code for an empty list.

456a ⟨*other methods of class* `List` 456a⟩≡ (455) 456b▷

```
(method addLast:  (item)   (insertAfter: (pred sentinel) item))
(method addFirst: (item)   (insertAfter: sentinel item))
(method add: (item)        (addLast: self item))
```

To remove the first element of a list, we remove the element after the sentinel.

456b ⟨*other methods of class* `List` 456a⟩+≡ (455) ◁456a 456c▷

```
(method removeFirst ()     (deleteAfter sentinel))
```

Removing the last element is left as Exercise 11.

To remove a particular element, we delegate the job to the first cons cell in the list.

456c ⟨*other methods of class* `List` 456a⟩+≡ (455) ◁456b 456d▷

```
(method remove:ifAbsent: (oldObject exnBlock)
    (rejectOne:ifAbsent:withPred:
        (cdr sentinel)
        (block (x) (= oldObject (car x)))
        exnBlock
        sentinel))
```

Regarded as a `SequenceableCollection`, a `List` object must be indexable by integers; indexing starts at 1. This fact is reflected in the definitions of `firstKey`, `lastKey`, and `at:put:`.

456d ⟨*other methods of class* `List` 456a⟩+≡ (455) ◁456c

```
(method firstKey () 1)
(method lastKey  () (size self))
(method at:put: (n value) (locals tmp)
    (set tmp (cdr sentinel))
    (whileFalse: [(= n 1)]
       [(set n (- n 1))
        (set tmp (cdr tmp))])
    (car: tmp value))
```

`at:put:` simply walks the `car` links `n` − 1 times. It provides no protection against `n`'s being out of range. You can protect against out-of-range errors in more than one way; Exercise 13 asks you to build and compare two protective mechanisms.

The hard work of implementing the list method is done by methods defined on cons cells. We begin by showing the simple methods that expose the representation as a pair of car and cdr. The `pred:` method makes it possible to tell *any* cons cell what its predecessor is, but unless the cons cell is a sentinel, the information is thrown away. (The sentinel is a special subclass of `Cons`.)

456e ⟨*classes that define cons cells and sentinels* 456e⟩≡ (455) 458a▷

```
(class Cons Object
    (car cdr)
    (method car ()           car)
    (method cdr ()           cdr)
    (method car: (anObject)  (set car anObject) self)
    (method cdr: (anObject)  (set cdr anObject) self)
    (method pred: (aCons)    nil)
    ⟨more methods of class Cons 457a⟩
)
```

Insertion and deletion use the standard pointer manipulations for a circularly linked list. The only unusual bit is that when we delete or insert a node, we notify the successor node of its new predecessor.

457a ⟨*more methods of class* Cons 457a⟩≡ (456e) 457b▷

```
(method deleteAfter () (locals answer)
    (set answer (car cdr))
    (set cdr    (cdr cdr))
    (pred: cdr self)
    answer)
(method insertAfter: (anObject)
    (set cdr (car: (cdr: (new Cons) cdr) anObject))
    (pred: (cdr cdr) cdr)
    anObject)
```

Iteration and removal really take advantage of object-oriented programming. By defining one method differently on class `Cons` and on class `ListSentinel`, we create code that iterates over a list without ever using an explicit `if` or `while`—all it does is method dispatch. To make the computation a little clearer, I present some of the methods of class `Cons` right next to the corresponding methods of class `ListSentinel`.

To iterate over all the cons cells in a list, we do the current cons cell, then continue with a tail call to the `do:` method of the next cons cell. The iteration terminates in the sentinel, whose `do:` method does nothing.

457b ⟨*more methods of class* Cons 457a⟩+≡ (456e) ◁457a 457d▷

```
(method do: (aBlock)
    (value aBlock car)
    (do: cdr aBlock))
```

457c ⟨*iterating methods of class* ListSentinel 457c⟩≡ (458a) 457e▷

```
(method do: (aBlock) nil)
```

Similarly, to remove the first cons cell satisfying `aBlock`, we try the current cons cell, and if it works, we remove it using `(deleteAfter pred)`. Otherwise, we continue by trying the next cons cell. If we get to the sentinel, we haven't found what we're looking for, and we terminate the loop by sending the `value` message to the exception block.

457d ⟨*more methods of class* Cons 457a⟩+≡ (456e) ◁457b

```
(method rejectOne:ifAbsent:withPred: (aBlock exnBlock pred)
    (if (value aBlock self)
        [(deleteAfter pred)]
        [(rejectOne:ifAbsent:withPred: cdr aBlock exnBlock self)]))
```

457e ⟨*iterating methods of class* ListSentinel 457c⟩+≡ (458a) ◁457c

```
(method rejectOne:ifAbsent:withPred: (aBlock exnBlock pred)
    (value exnBlock))
```

The remaining instead methods of class `ListSentinel` provide access to the `pred` pointer. The class method `new` allocates a new sentinel, whose `pred` and `cdr` both point to itself, which represents an empty list.

458a ⟨*classes that define cons cells and sentinels* 456e⟩+≡ (455) ◁456e

```
(class ListSentinel Cons
    (pred)
    (method pred: (aCons)   (set pred aCons))
    (method pred  ()        pred)
    (class-method new ()    (locals tmp)
        (set tmp (new super))
        (pred: tmp tmp)
        (cdr:  tmp tmp)
        tmp)
    ⟨iterating methods of class ListSentinel 457c⟩)
```

Array

In Smalltalk, arrays are one-dimensional and have a fixed size. Elements are indexed with integer keys, starting from 1. Many of `Array`'s methods are primitive, including array creation and the `at:`, `at:put:`, and `size` methods. These methods are defined in the interpreter, in chunks 482b–482e in Section 10.6.5.

458b ⟨*collection classes* 448b⟩+≡ (495d) ◁455

```
(class Array SequenceableCollection
    () ; representation is primitive
    (class-method new: primitive arrayNew:)
    (class-method new () (error: self #size-of-Array-must-be-specified))
    (method size      primitive arraySize)
    (method at:       primitive arrayAt:)
    (method at:put:   primitive arrayAt:Put:)
    (method species   () Array)
    (method printName () nil) ; names of arrays aren't printed
    ⟨other methods of class Array 458c⟩
)
```


Since it's not useful to create an array without giving a size, we redefine class method `new` so that it reports an error.

An array is mutable, but it has a fixed size, so trying to add or remove an element is senseless.

⟨*other methods of class* `Array` 458c⟩≡ (458b) 458d▷

```
(method add:             (x)   (fixedSizeError self))
(method remove:ifAbsent: (x b) (fixedSizeError self))
(method fixedSizeError   ()    (error: self #arrays-have-fixed-size))
```

Because `add:` doesn't work, the inherited implementations of `select:` and `collect:` won't work either.

458d ⟨*other methods of class* `Array` 458c⟩+≡ (458b) ◁458c 458e▷

```
(method select:  (_) (error: self #select-on-arrays-not-implemented))
(method collect: (_) (error: self #collect-on-arrays-not-implemented))
```

Like lists, arrays have keys from 1 to size. We iterate over the keys.

458e ⟨*other methods of class* `Array` 458c⟩+≡ (458b) ◁458d

```
(method firstKey () 1)
(method lastKey  () (size self))
(method do: (aBlock) (locals index)
    (set index (firstKey self))
    (timesRepeat: (size self)
       [(value aBlock (at: self index))
        (set index (+ index 1))]))
```

10.5.4 Magnitudes and numbers

Magnitudes can be ordered, but that's it. In particular, you can't do arithmetic on magnitudes. Moreover, the rules of Smalltalk-80 require that magnitudes not use the built-in equality (object identity). μSmalltalk enforces those rules by redefining the = method to be a subclass's responsibility. A subclass's only other responsibility is the < method; the other methods in the `Magnitude` protocol are all implemented in class `Magnitude`.

459a ⟨*numeric classes* 459a⟩≡ (495d) 459b▷

```
(class Magnitude Object
    () ; abstract class
    (method =  (x) (subclassResponsibility self)) ; may not inherit from Object
    (method <  (x) (subclassResponsibility self))
    (method >  (y) (< y self))
    (method <= (x) (not (> self x)))
    (method >= (x) (not (< self x)))
    (method min: (aMagnitude) (if (< self aMagnitude) [self] [aMagnitude]))
    (method max: (aMagnitude) (if (> self aMagnitude) [self] [aMagnitude]))
)
```

If an object supports not just comparison but also arithmetic, it's a number. `Number` is also an abstract class; any subclass must supply four arithmetic operations and four conversions, as well as the = and < methods required by the `Magnitude` protocol.

459b ⟨*numeric classes* 459a⟩+≡ (495d) ◁459a 460c▷

```
(class Number Magnitude
    ()  ; abstract class
    ;;;;;;; basic Number protocol
    (method +   (aNumber)     (subclassResponsibility self))
    (method *   (aNumber)     (subclassResponsibility self))
    (method negated    ()     (subclassResponsibility self))
    (method reciprocal ()     (subclassResponsibility self))

    (method asInteger  ()     (subclassResponsibility self))
    (method asFraction ()     (subclassResponsibility self))
    (method asFloat    ()     (subclassResponsibility self))
    (method coerce: (aNumber) (subclassResponsibility self))
    ⟨other methods of class Number 459c⟩
)
```

The remaining methods of class `Number` define a double handful of operations that can be reused by all numbers. First, from the inverse operations `negated` and `reciprocal`, we get binary subtraction, absolute value, and division.

459c ⟨*other methods of class* `Number` 459c⟩≡ (459b) 459d▷

```
(method -   (y) (+ self (negated  y)))
(method abs ()  (if (negative self) [(negated  self)] [self]))
(method /   (y) (* self (reciprocal y)))
```

The sign predicates, like `negative`, are implemented by comparing `self` with zero. The implementations also show how to use the `coerce:` method required by the protocol in Figure 10.12 on page 424.

459d ⟨*other methods of class* `Number` 459c⟩+≡ (459b) ◁459c 460a▷

```
(method negative         () (<  self (coerce: self 0)))
(method positive         () (>= self (coerce: self 0)))
(method strictlyPositive () (>  self (coerce: self 0)))
```

The implementation of `squared` is easy. The implementation of `raisedToInteger:` computes x^n using a standard algorithm that requires $O(\log n)$ multiplications. The algorithm has two base cases, for x^0 and x^1.

460a ⟨*other methods of class* Number 459c⟩+≡ (459b) ◁459d 460b▷

```
(method squared () (* self self))
(method raisedToInteger: (anInteger)
    (if (= anInteger 0)
        [(coerce: self 1)]
        [(if (= anInteger 1) [self]
            [(* (squared (raisedToInteger: self (div: anInteger 2)))
                (raisedToInteger: self (mod: anInteger 2)))])]))
```

Our implementation of square root uses Newton-Raphson iteration. For input n, the algorithm works by assuming the initial approximation $x_0 = 1$ and improving it at step i, $i \geq 0$, by computing $x_{i+1} = (x_i + n/x_i)/2$. We need some *convergence condition*: a predicate on x_i and x_{i-1} indicating when they are close enough to accept x_i as the answer.[11] We stop iterating when $|x_i - x_{i-1}| < \epsilon$. Using `coerce:` ensures we can use the same `sqrt` method for both fractions and floats.

460b ⟨*other methods of class* Number 459c⟩+≡ (459b) ◁460a

```
(method sqrt () (sqrtWithin: self (coerce: self (/ 1 10))))
(method sqrtWithin: (epsilon) (locals two x_{i-1} x_{i})
    ; find square root of receiver within epsilon
    (set two     (coerce: self 2))
    (set x_{i-1} (coerce: self 1))
    (set x_{i}   (/ (+ x_{i-1} (/ self x_{i-1})) two))
    (while [(> (abs (- x_{i-1} x_{i})) epsilon)]
           [(set x_{i-1} x_{i})
            (set x_{i} (/ (+ x_{i-1} (/ self x_{i-1})) two))])
    x_{i})
```


Integers

`Integer` is an abstract class. A full Smalltalk-80 system defines three concrete subclasses of `Integer`: `SmallInteger`, for integers that fit comfortably in a single machine word; `LargePositiveInteger`, for arbitrarily large positive integers; and `LargeNegativeInteger`, for arbitrarily large negative integers. μSmalltalk defines only `SmallInteger`, but Exercise 32 explains how you can add large positive and negative integers.

Integer division and modulus are defined only on integers, as are greatest common denominator and least common multiple.

⟨*numeric classes* 459a⟩+≡ (495d) ◁459b 461e▷

```
(class Integer Number
    () ; abstract class
    (method div: (n) (subclassResponsibility self))
    (method mod: (n) (- self (* n (div: self n))))
    (method gcd: (n) (if (= n (coerce: self 0)) [self] [(gcd: n (mod: self n))]))
    (method lcm: (n) (* self (div: n (gcd: self n))))
    ⟨other methods of class Integer 461a⟩
)
```

[11] The idea is that if $x_i \approx x_{i-1}$, $x_i = (x_{i-1} + n/x_{i-1})/2 \approx (x_i + n/x_i)/2$, and solving yields $x_i \approx \sqrt{n}$.

To understand how the numeric classes interoperate, focus your attention on the conversion methods. Each class of number needs to know how to convert itself to every other class of number. Each class therefore implements the `asFraction`, `asFloat`, and `asInteger` methods. To implement these methods, a class has to know not only its own representation and methods, but what methods of the other numeric classes are needed to create a number. For example, the `Integer` class needs to know what methods of classes `Fraction` and `Float` to use to to create an equivalent `Fraction` or `Float`.

To create a fraction, use method `num:den:` with a numerator and a denominator; to create a float, use method `mant:exp:` with a mantissa and an exponent.

461a ⟨*other methods of class* `Integer` 461a⟩≡ (460c) 461b▷
```
(method asFraction () (num:den:  Fraction self 1))
(method asFloat    () (mant:exp: Float    self 0))
(method asInteger  () self)
```

Just as it's up to `Integer` to know how to convert itself to another kind of number, it's up to the other kinds of numbers to know how to convert themselves to integer. Thus, the `coerce:` method on class `Integer` simply delegates to the `asInteger` method of the argument.

461b ⟨*other methods of class* `Integer` 461a⟩+≡ (460c) ◁461a 461c▷
```
(method coerce: (aNumber) (asInteger aNumber))
```

When integers are divided, the result isn't an integer; it's a fraction.

461c ⟨*other methods of class* `Integer` 461a⟩+≡ (460c) ◁461b 461d▷
```
(method reciprocal () (num:den: Fraction 1 self))
(method / (aNumber) (/ (asFraction self) aNumber))
```

The integer method `timesRepeat:` executes a loop a finite number of times.

461d ⟨*other methods of class* `Integer` 461a⟩+≡ (460c) ◁461c
```
(method timesRepeat: (aBlock) (locals count)
    (ifTrue: (negative self) [(error: self #negative-repeat-count)])
    (set count self)
    (while [(!= count 0)]
         [(value aBlock)
          (set count (- count 1))]))
```

`SmallInteger` is the only implementation of integers built into μSmalltalk. Almost all its methods are primitive. They are defined in chunks 480f–481c.

461e ⟨*numeric classes* 459a⟩+≡ (495d) ◁460c 462a▷
```
(class SmallInteger Integer
    () ; primitive representation
    (class-method new: primitive newSmallInteger:)
    (class-method new   () (new: self 0))
    (method negated     () (- 0 self))
    (method print primitive printSmallInteger)
    (method +     primitive +)
    (method -     primitive -)
    (method *     primitive *)
    (method div:  primitive div)
    (method =     primitive eqObject)
    (method <     primitive <)
    (method >     primitive >)
)
```

These methods are not good enough to support *mixed arithmetic*, e.g., comparison of integers and fractions. Writing better methods is a task you can do in Exercise 29.

Fractions

Class `Fraction` represents a rational number as a fraction (numerator and denominator). As a representation invariant, we keep each fraction in reduced form, meaning that (1) the numerator and denominator are relatively prime; (2) if the numerator is zero, the denominator is one; and (3) the denominator is positive. We use reduced form because when two fractions are in reduced form, it's easy to tell if they are equal. The private methods `divReduce` and `signReduce` put a fraction into reduced form.

462a ⟨*numeric classes* 459a⟩+≡ (495d) ◁461e 464d▷

```
(class Fraction Number
    (num den)
    (class-method num:den: (a b) (initNum:den: (new self) a b))
    (method initNum:den: (a b) ; private
        (setNum:den: self a b)
        (signReduce self)
        (divReduce self))
    (method setNum:den: (a b) (set num a) (set den b) self) ; private
    ⟨other methods of class Fraction 462b⟩
)
```

Private method `setNum:den:` sets the numerator and denominator of a fraction without reducing it.

Methods `signReduce` and `divReduce` establish different parts of the representation invariant; `signReduce` ensures that the denominator is positive, and `divReduce` ensures both that zero is properly represented and that the numerator and denominator have no common divisors. Only `initNum:den:` calls both, since it is the only method whose arguments may be arbitrary integers. The other arithmetic operations call at most one, depending upon what kind of reduction might be needed for the result of that operation. For example, if `negate`'s receiver is in reduced form, so is its result, and therefore `negate` needn't call `signReduce` or `divReduce`. As another example, if two fractions in reduced form are added, the result's denominator is not negative, because it is the product of two nonnegative denominators, but the result's numerator and denominator may have common factors, so the `+` method calls `divReduce`. A similar argument applies to `*`.

if 446
ifTrue:,
in class Boolean 446
in class False 447c
in class True 447b
negative,
in class Float 466c
in class Fraction 464c
in class Large-Negative-Integer 819b
in class Large-Positive-Integer 819a
in class Number 459d
Number 459b

⟨*other methods of class* `Fraction` 462b⟩≡ (462a) 463a▷

```
(method signReduce () ; private
    (ifTrue: (negative den)
        [(set num (negated num)) (set den (negated den))])
    self)

(method divReduce () (locals temp) ; private
    (if (= num 0)
        [(set den 1)]
        [(set temp (gcd: (abs num) den))
         (set num  (div: num temp))
         (set den  (div: den temp))])
    self)
```

Like method `setNum:den:`, methods `signReduce` and `divReduce` answer `self`. This convention, which is common in object-oriented programming, simplifies the implementations of the arithmetic operations.

A binary arithmetic operation needs access to the numerator and denominator of its argument as well as the receiver. This access is provided by the private methods `num` and `den`.

463a ⟨*other methods of class* `Fraction` 462b⟩+≡ (462a) ◁462b 463b▷
```
(method num () num)
(method den () den)
```

We begin our development of the binary operations with the `=` and `<` methods that are required by the `Magnitude` protocol. Because fractions are in reduced form, two fractions are equal if and only if their numerators and denominators are equal; no arithmetic is required. The `<` comparison requires multiplication; $\frac{n}{d} < \frac{n'}{d'}$ if and only if $nd' < n'd$.

463b ⟨*other methods of class* `Fraction` 462b⟩+≡ (462a) ◁463a 463c▷
```
(method = (f)
    (and: (= num (num f)) [(= den (den f))]))
(method < (f)
    (< (* num (den f)) (* (num f) den)))
```

Negation simply negates the numerator. Because numbers are *immutable*, we return a new fraction with the negated numerator.

463c ⟨*other methods of class* `Fraction` 462b⟩+≡ (462a) ◁463b 463d▷
```
(method negated () (setNum:den: (new Fraction) (negated num) den))
```

Instead of simply calling `(num:den: Fraction (negated num) den)`, we call `setNum:den:` on a new fraction. By avoiding `num:den:`, we avoid unnecessary calls to `divReduce` and `signReduce`, doing only what is needed to maintain the representation invariant.

Multiplication uses the algebraic law $\frac{n}{d}\frac{n'}{d'} = \frac{nn'}{dd'}$. The numerator nn' and denominator dd' can have common factors, so `divReduce` is necessary, but dd' cannot be negative, so `signReduce` can be omitted.

463d ⟨*other methods of class* `Fraction` 462b⟩+≡ (462a) ◁463c 463e▷
```
(method * (f)
    (divReduce
        (setNum:den: (new Fraction)
                     (* num (num f))
                     (* den (den f)))))
```

The simplest implementation of `+` would use the algebraic law $\frac{n}{d} + \frac{n'}{d'} = \frac{nd''+n'd}{dd'}$, then call `divReduce` to eliminate common factors (e.g., $\frac{1}{2} + \frac{1}{2} = \frac{4}{4}$). We reduce the possibility of overflow by letting `temp` $= \mathrm{lcm}(d, d')$, putting `temp` in the denominator, and using $\frac{\texttt{temp}}{d}$ and $\frac{\texttt{temp}}{d'}$ as needed. This tweak keeps the square-root computations in Section 10.3.5 from overflowing. If needed, a similar technique could be applied to multiplication.

463e ⟨*other methods of class* `Fraction` 462b⟩+≡ (462a) ◁463d 463f▷
```
(method + (f) (locals temp)
    (set temp (lcm: den (den f)))
    (divReduce
        (setNum:den: (new Fraction)
                     (+ (* num (div: temp den)) (* (num f) (div: temp (den f))))
                     temp)))
```

The method `reciprocal` inverts a fraction. This operation leaves a positive fraction in reduced form, but it leaves a negative fraction with a negative denominator. Sending `signReduce` to the inverted fraction sets things right regardless of the original sign. (The reciprocal of zero cannot be put into reduced form, but nothing can be done about it.)

463f ⟨*other methods of class* `Fraction` 462b⟩+≡ (462a) ◁463e 464a▷
```
(method reciprocal ()
  (signReduce (setNum:den: (new Fraction) den num)))
```

The print method for fractions shows numerator and denominator.

⟨*other methods of class* Fraction 462b⟩+≡ (462a) ◁463f 464b▷
```
(method print () (print num) (print #/) (print den) self)
```

Here are the conversions. Like an integer, a fraction knows how to convert itself to a value of any other numeric class. The knowledge of how to do the inverse conversion, from a value of another numeric class to a fraction, lies the other class, in its implementation of asFraction.

⟨*other methods of class* Fraction 462b⟩+≡ (462a) ◁464a 464c▷
```
(method asInteger  () (div: num den))
(method asFloat    () (/ (asFloat num) (asFloat den)))
(method asFraction () self)
(method coerce: (aNumber) (asFraction aNumber))
```

Converting a fraction to an integer may lose precision.

Finally, the sign methods can be implemented by looking only at a fraction's numerator, so instead of inheriting the default implementations from Number, we redefine those methods.

⟨*other methods of class* Fraction 462b⟩+≡ (462a) ◁464b
```
(method negative         () (negative num))
(method positive         () (positive num))
(method strictlyPositive () (strictlyPositive num))
```

Floating-point numbers

The original Smalltalk systems were built on the Xerox Alto, the world's first personal computer. Because the Alto had no hardware support for floating-point computation, floating-point computations were done in software. The implementation we present here would be suitable for such a machine (although more bits of precision in the mantissa would be welcome).

A number in floating-point form is represented by a mantissa and exponent (of 10). That is, the Float having mantissa m and exponent e represents $m \times 10^e$; m and e can be negative. As a representation invariant, we keep the absolute value of the mantissa at most $2^{15} - 1$. This restriction ensures that we can multiply two mantissas without overflow, even on an implementation that provides only 31-bit small integers.[12] When a mantissa's magnitude exceeds $2^{15} - 1$, the private normalize method divides the mantissa by 10 and increments the exponent until the mantissa is small enough.

⟨*numeric classes* 459a⟩+≡ (495d) ◁462a
```
(class Float Number
    (mant exp)
    (class-method mant:exp: (m e) (initMant:exp: (new self) m e))
    (method initMant:exp: (m e) ; private
        (set mant m) (set exp e) (normalize self))
    (method normalize ()    ; private
        (while [(> (abs mant) 32767)]
               [(set mant (div: mant 10))
                (set exp (+ exp 1))])
        self)
    ⟨other methods of class Float 465a⟩
)
```

[12] Some implementations of ML reserve 1 bit as a dynamic-type tag or as a tag for the garbage collector.

The private methods of `mant` and `exp` provide access to the mantissas and exponents of other floats.

465a ⟨*other methods of class* `Float` 465a⟩≡ (464d) 465b▷
```
(method mant () mant)   ; private
(method exp  () exp)    ; private
```

Comparing two floats with different exponents is awkward, so to compare two values, we take their difference and compare it with zero. We add a method `isZero` for this purpose.

465b ⟨*other methods of class* `Float` 465a⟩+≡ (464d) ◁465a 465c▷
```
(method < (x) (negative (- self x)))
(method = (x) (isZero   (- self x)))
(method isZero () (= mant 0))
```

Negation is easy: we answer a new float with a mantissa of the opposite sign.

465c ⟨*other methods of class* `Float` 465a⟩+≡ (464d) ◁465b 465d▷
```
(method negated () (mant:exp: Float (negated mant) exp))
```

Method `negated`, together with the `+` method, also supports subtraction and comparison. Because of the way methods are inherited and work with one another, all the programming effort required to add, subtract, or compare floating-point numbers with different exponents can be concentrated in the `+` method. This design represents another victory for inheritance.

The implementation of `+` is based on the algebraic law $m \times 10^e = (m \times 10^{e-e'}) \times 10^{e'}$. This law implies

$$m \times 10^e + m' \times 10^{e'} = (m \times 10^{e-e'} + m') \times 10^{e'}.$$

We provide a naïve implementation which enforces $e - e' \geq 0$. This implementation risks overflow, but at least overflow is something that can be detected. A naïve implementation using $e - e' \leq 0$ might well lose valuable bits of precision from m. A better implementation can be constructed using the ideas in Exercise 28.

465d ⟨*other methods of class* `Float` 465a⟩+≡ (464d) ◁465c 465e▷
```
(method + (prime)
    (if (>= exp (exp prime))
        [(mant:exp: Float (+ (* mant (raisedToInteger: 10 (- exp (exp prime))))
                             (mant prime))
                          (exp prime))]
        [(+ prime self)]))
```

Multiplication is much simpler: $(m \times 10^e) \times (m' \times 10^{e'}) = (m \times m') \times 10^{e+e'}$. The product's mantissa $m \times m'$ may be large, but the class method `mant:exp:` normalizes it.

465e ⟨*other methods of class* `Float` 465a⟩+≡ (464d) ◁465d 465f▷
```
(method * (prime)
    (mant:exp: Float (* mant (mant prime)) (+ exp (exp prime))))
```

We compute the reciprocal using the algebraic law

$$\frac{1}{m \times 10^e} = \frac{10^9}{m \times 10^9 \times 10^e} = \frac{10^9}{m} \times 10^{-e-9}.$$

This technique ensures we don't lose too much precision.

465f ⟨*other methods of class* `Float` 465a⟩+≡ (464d) ◁465e 465g▷
```
(method reciprocal ()
    (mant:exp: Float (div: 1000000000 mant) (- -9 exp)))
```

Coercing converts to `Float`, and converting `Float` to `Float` is the identity.

465g ⟨*other methods of class* `Float` 465a⟩+≡ (464d) ◁465f 466a▷
```
(method coerce: (aNumber) (asFloat aNumber))
(method asFloat () self)
```

When converting a float to another class of number, a negative exponent means divide, and a positive exponent means multiply.

⟨*other methods of class* `Float` 465a⟩+≡ (464d) ◁465g 466b▷

```
(method asInteger ()
    (if (< exp 0)
        [(div: mant (raisedToInteger: 10 (negated exp)))]
        [(*    mant (raisedToInteger: 10 exp))]))
```

To get a fraction, we either put a power of 10 in the denominator, or we make a product with 1 in the denominator.

⟨*other methods of class* `Float` 465a⟩+≡ (464d) ◁466a 466c▷

```
(method asFraction ()
    (if (< exp 0)
        [(num:den: Fraction mant (raisedToInteger: 10 (negated exp)))]
        [(num:den: Fraction (* mant (raisedToInteger: 10 exp)) 1)]))
```

The sign tests aren't just an optimization, as they were with `Fraction`; `negative` is required by the implementation of `<`.

⟨*other methods of class* `Float` 465a⟩+≡ (464d) ◁466b 466d▷

```
(method negative         () (negative mant))
(method positive         () (positive mant))
(method strictlyPositive () (strictlyPositive mant))
```

We print floating-point numbers as $m \times 10^e$. But we want to avoid printing a number like 77 as 770×10^{-1}. So if the `print` method sees a number with a negative exponent and a mantissa that is a multiple of 10, it divides the mantissa by 10 and increases the exponent, continuing until the exponent reaches zero or the mantissa is no longer a multiple of 10. As a result, a whole number always prints as a whole number times 10^0, no matter what its internal representation is.

⟨*other methods of class* `Float` 465a⟩+≡ (464d) ◁466c

```
(method print ()
    (print-normalize self)
    (print mant) (print #x10^) (print exp)
    (normalize self))

(method print-normalize ()
    (while [(and: (< exp 0) [(= (mod: mant 10) 0)])]
        [(set exp (+ exp 1)) (set mant (div: mant 10))]))
```

10.6 Interpreter and operational semantics

This section presents the abstract syntax, operational semantics, and implementation of μSmalltalk.

10.6.1 Abstract syntax and values

The abstract-syntax constructors VAR, SET, SEND, BEGIN, and BLOCK resemble the kinds of abstract syntax you have seen in other interpreters. Three other constructors solve problems that are unique to Smalltalk.

- Literal values require special treatment. A literal value must ultimately stand for an object, and the object must have a class, but while the interpreter is being bootstrapped, the classes aren't yet defined (Section 10.6.4). So for example, we can't create a complete representation of a literal integer until the class `Integer` is defined.

  To solve this problem, we handle literals somewhat differently than in other interpreters. The LITERAL node doesn't contain a complete value; it contains only a *representation.* Before the representation becomes a full-fledged value, it must be associated with a class.

- Once we do have a full-fledged value, we put it in a VALUE node, which includes both a class and a represention.

- The SUPER node makes it easy to recognize messages to **super** and give them the proper semantics.

467a ⟨*abstract syntax and values* 467a⟩≡ (475a) 467b▷

```
datatype exp = VAR     of name
             | SET     of name * exp
             | SEND    of srcloc * name * exp * exp list
             | BEGIN   of exp list
             | BLOCK   of name list * exp list
             | LITERAL of rep
             | VALUE   of value
             | SUPER
```

Unlike other interpreters in this book, the μSmalltalk interpreter keeps track of source-code locations. The srcloc field in the SEND node is used in diagnostic error messages.

In addition to our old friends VAL, EXP, and USE, a definition may be a block definition (DEFINE) or a class definition (CLASSD).

467b ⟨*abstract syntax and values* 467a⟩+≡ (475a) ◁467a 468a▷

```
and def
  = VAL    of name * exp
  | EXP    of exp
  | DEFINE of name * name list * exp
  | CLASSD of { name    : string
              , super   : string
              , ivars   : string list     (* instance variables *)
              , methods : (method_kind * name * method_impl) list
              }
  | USE of name
and method_kind = IMETHOD          (* instance method *)
                | CMETHOD          (* class method    *)
and method_impl = USER_IMPL of name list * name list * exp
                | PRIM_IMPL of name
```

type name 214
type rep 468a
type value 468d

In a compiled or bytecoded implementation of Smalltalk, the representation of an object is a sequence of machine words. The methods, and in particular the primitive methods, "know" which words stand for integers, which words are slots for instance variables, and so on. But our interpreter is written in ML, which doesn't provide direct access to machine words. In our implementation, we define a `rep` type for representations. The representation of a user-defined object is an environment giving the locations of instance variables. Representations of arrays, numbers, symbols, blocks, and classes are primitive.

468a ⟨*abstract syntax and values* 467a⟩+≡ (475a) ◁467b 468b▷

```
and    rep = USER    of value ref env (* collection of named instance variables *)
           | ARRAY   of value Array.array
           | NUM     of int
           | SYM     of name
           | CLOSURE of name list * exp list * value ref env * class
           | CLASSREP of class
```

A `CLOSURE` is a representation of a block; it captures not only the environment used to bind variables, but also the class used to interpret messages to `super`.

The representation of a class includes its superclass, instance variables, methods.

468b ⟨*abstract syntax and values* 467a⟩+≡ (475a) ◁468a 468c▷

```
and class  = CLASS of { name    : name           (* name of the class *)
                      , super   : class option   (* superclass, if any *)
                      , ivars   : string list    (* instance variables *)
                      , methods : method env     (* both exported and private *)
                      , id      : int            (* uniquely identifies class *)
                      }
```

Except for the distinguished root class, `Object`, every class has a superclass. A class's `ivars` and `methods` lists include only the instance variables and methods defined in that class, not those of its superclass.

Every class also has a unique identifier `id`. The unique identifier makes it easy to tell when two class objects are identical, as well as to implement methods like `isMemberOf:`, which tells us exactly what class an object is a member of. (See also the discussion of function `eqRep` in chunk 479b.)

μSmalltalk has two kinds of methods. Primitive methods are represented as ML functions; user-defined methods are represented as abstract syntax, which includes parameters, local variables, and a body. In each user-defined method, we also store the superclass of the class in which the method is defined, which we use to interpret messages sent to `super` from within that method.

468c ⟨*abstract syntax and values* 467a⟩+≡ (475a) ◁468b 468d▷

```
and method
  = PRIM_METHOD of name * (value * value list -> value)
  | USER_METHOD of { name : name, formals : name list, locals : name list, body : exp
                   , superclass : class (* used to send messages to super *)
                   }
```

type env 214
type exp 467a
type name 214

Finally, a value is a combination of class and representation. The representation is inherently *either* a primitive representation *or* a collection of instance variables, not a combination of both; it is therefore not useful to inherit from a class with a primitive representation.

468d ⟨*abstract syntax and values* 467a⟩+≡ (475a) ◁468c 469▷

```
withtype value = class * rep
```

As in the implementation of μScheme, a primitive method might raise the `RuntimeError` exception. Because the μSmalltalk interpreter uses more mutable state than other interpreters and is therefore harder to get right, we also define the exception `InternalError`, which indicates a bug in the interpreter. Raising `InternalError` is the moral equivalent of an assertion failure in a language like C.

469 ⟨*abstract syntax and values* 467a⟩+≡ (475a) ◁468d 475b▷

```
exception RuntimeError  of string (* error caused by user *)
exception InternalError of string (* bug in the interpreter *)
```

10.6.2 Operational semantics

The most unusual feature of μSmalltalk's operational semantics is that during the execution of a program, the behavior of a literal integer, symbol, or array can change. Such a change occurs if a μSmalltalk programmer redefines classes `SmallInteger`, `Symbol`, or `Array`. For example, if you complete Exercises 29, 32, and 33, you will change the behavior of integer literals to provide a seamless, transparent blend of small- and large-integer arithmetic, all without touching the μSmalltalk interpreter. The power comes at a price: if you make a mistake redefining `SmallInteger`, for example, you could render your interpreter unusable.

Expressions

In μSmalltalk, each expression is evaluated as part of a message send. The context of the message send determines the meanings of names: globals, instance variables, formal parameters, and local variables (Section 10.2.3). In μScheme, we could bind all those variables in a single environment. μSmalltalk is more dynamic than μScheme; it is permissible for a method to refer to a global variable bound only after the method is defined. We therefore use two environments: ξ, which binds the global variables, and ρ, which binds everything else. The environment ξ represents global state, and it is threaded through the interpreter. Environment ρ represents state that is private to objects and methods. When we evalute a block and create a closure, we capture only ρ, not ξ.

We need one more piece of context. In order to evaluate a message to `super`, we have to know the *static* superclass of the method definition within which we are currently evaluating. This class is *not* the same as the superclass of the object that received the message; see Exercise 38. We write this static superclass as $c_{\texttt{super}}$.

The judgment for evaluation of an expression has the form $\langle e, \rho, c_{\texttt{super}}, \xi, \sigma\rangle \Downarrow \langle v, \sigma'\rangle$. The form of this judgment shows that evaluating an expression may change the store σ, but it does not change any environment.

Variables and assignment As in Impcore, we have environments ρ and ξ for local and global variables. Both environments bind names to mutable locations.

$$\frac{x \in \operatorname{dom}\rho}{\langle \textsc{var}(x), \rho, c_{\texttt{super}}, \xi, \sigma\rangle \Downarrow \langle \sigma(\rho(x)), \sigma\rangle} \qquad (\textsc{Var})$$

$$\frac{x \notin \operatorname{dom}\rho \qquad x \in \operatorname{dom}\xi}{\langle \textsc{var}(x), \rho, c_{\texttt{super}}, \xi, \sigma\rangle \Downarrow \langle \sigma(\xi(x)), \sigma\rangle} \qquad (\textsc{GlobalVar})$$

Assignment translates the name into a location, then changes the value in that location. As in μScheme, we thread the store.

$$\frac{x \in \mathrm{dom}\,\rho \qquad \rho(x) = \ell \qquad \langle e, \rho, c_{\texttt{super}}, \xi, \sigma\rangle \Downarrow \langle v, \sigma'\rangle}{\langle \mathrm{SET}(x, e), \rho, c_{\texttt{super}}, \xi, \sigma\rangle \Downarrow \langle v, \sigma'\{\ell \mapsto v\}\rangle} \quad \text{(ASSIGN)}$$

$$\frac{x \notin \mathrm{dom}\,\rho \qquad x \in \mathrm{dom}\,\xi \qquad \xi(x) = \ell \qquad \langle e, \rho, c_{\texttt{super}}, \xi, \sigma\rangle \Downarrow \langle v, \sigma'\rangle}{\langle \mathrm{SET}(x, e), \rho, c_{\texttt{super}}, \xi, \sigma\rangle \Downarrow \langle v, \sigma'\{\ell \mapsto v\}\rangle} \quad \text{(ASSIGNGLOBAL)}$$

Self and super In μSmalltalk, `self` is treated as an ordinary variable, so one of the variable rules applies. In most contexts, `super` behaves exactly like `self`.

$$\frac{\langle \mathrm{VAR}(\texttt{self}), \rho, c_{\texttt{super}}, \xi, \sigma\rangle \Downarrow \langle v, \sigma\rangle}{\langle \mathrm{SUPER}, \rho, c_{\texttt{super}}, \xi, \sigma\rangle \Downarrow \langle v, \sigma\rangle} \quad \text{(SUPER)}$$

When a message is sent to `super`, it behaves differently from `self`; the details are in the rules for message send, which appear on pages 471–472.

Values As in Impcore, a VALUE node evaluates to itself without changing the store.

$$\frac{}{\langle \mathrm{VALUE}(v), \rho, c_{\texttt{super}}, \xi, \sigma\rangle \Downarrow \langle v, \sigma\rangle} \quad \text{(VALUE)}$$

Literals As noted above, a literal's behavior depends on the current definitions of `SmallInteger`, `Symbol`, and `Array`. A literal evaluates to a value, which as shown in the ML code above, has the form $\{\!|\, class, rep \,|\!\}$. We give formal semantics only for integer and symbol literals.

$$\frac{}{\langle \mathrm{LITERAL}(\mathrm{NUM}(n)), \rho, c_{\texttt{super}}, \xi, \sigma\rangle \Downarrow \langle \{\!|\, \sigma(\xi(\texttt{SmallInteger})), \mathrm{NUM}(n) \,|\!\}, \sigma\rangle} \quad \text{(LITERALNUMBER)}$$

$$\frac{}{\langle \mathrm{LITERAL}(\mathrm{SYM}(s)), \rho, c_{\texttt{super}}, \xi, \sigma\rangle \Downarrow \langle \{\!|\, \sigma(\xi(\texttt{Symbol})), \mathrm{SYM}(s) \,|\!\}, \sigma\rangle} \quad \text{(LITERALSYMBOL)}$$

The class of a literal number or a literal symbol is taken from the current global enviroment ξ, which means its behavior can be changed by redefining `SmallInteger` or `Symbol`.

Blocks A block is much like a `lambda` abstraction, except that the body *es* is a sequence of expressions, not a single expression. The block captures the current environment in a closure.

$$\frac{}{\langle \mathrm{BLOCK}(\langle x_1, \ldots, x_n\rangle, es), \rho, c_{\texttt{super}}, \xi, \sigma\rangle \Downarrow \langle \{\!|\, \sigma(\xi(\texttt{Block})), \mathrm{CLOSURE}(\langle x_1, \ldots, x_n\rangle, es, \rho, c_{\texttt{super}}) \,|\!\}, \sigma\rangle} \quad \text{(MKCLOSURE)}$$

Message send The rules for message passing are complex. To express the method-dispatch algorithm, we introduce the judgment form

$$\boxed{m \rhd c \mathbin{@} imp}$$

which should be pronounced "sending m to c is answered by *imp*." The judgment means that sending a message m to an object of class c dispatches to the implementation *imp*. The implementation may be a user-defined method or a primitive method. The idea is that the judgment $m \rhd c \mathbin{@} imp$ is provable if and only if *imp* is the *first* method named m defined either on class c or on one of c's superclasses. The formal semantics is left as Exercise 34.

The first rule for message-send describes what happens a message dispatches to a user-defined method.

$$\frac{\begin{array}{c} e \neq \text{SUPER} \qquad m \neq \text{VALUE} \\ \langle e, \rho, c_{\texttt{super}}, \xi, \sigma\rangle \Downarrow \langle\langle c, r\rangle, \sigma_0\rangle \\ \langle e_1, \rho, c_{\texttt{super}}, \xi, \sigma_0\rangle \Downarrow \langle v_1, \sigma_1\rangle \quad \cdots \quad \langle e_n, \rho, c_{\texttt{super}}, \xi, \sigma_{n-1}\rangle \Downarrow \langle v_n, \sigma_n\rangle \\ m \rhd c \mathbin{@} \text{USER\_METHOD}(\_, \langle x_1, \ldots, x_n\rangle, \langle y_1, \ldots, y_k\rangle, e_m, \mathtt{s}) \\ \ell_1, \ldots, \ell_n \notin \operatorname{dom}\sigma_n \qquad \ell'_1, \ldots, \ell'_k \notin \operatorname{dom}\sigma_n \\ \ell_1, \ldots, \ell_n, \ell'_1, \ldots, \ell'_k \text{ all distinct} \\ \hat{\sigma} = \sigma_n\{\ell_1 \mapsto v_1, \ldots \ell_n \mapsto v_n, \ell'_1 \mapsto \mathit{nil}, \ldots \ell'_k \mapsto \mathit{nil}\} \\ \rho' = \mathtt{instanceVars}(r) \\ \langle e_m, \rho'\{x_1 \mapsto \ell_1, \ldots, x_n \mapsto \ell_n, y_1 \mapsto \ell'_1, \ldots, y_k \mapsto \ell'_k\}, \mathtt{s}, \xi, \hat{\sigma}\rangle \Downarrow \langle v, \sigma'\rangle \end{array}}{\langle \text{SEND}(m, e, e_1, \ldots, e_n), \rho, c_{\texttt{super}}, \xi, \sigma\rangle \Downarrow \langle v, \sigma'\rangle} \qquad (\text{SENDUSER})$$

The rule has much in common with the closure rule for μScheme:

- The premises on the first line show that this is a rule for ordinary message send, not one of the special rules for sending VALUE or for sending messages to SUPER.
- The premises on the next two lines show the evaluation of the receiver e and of the actual parameters $e_1, \ldots, e_n$. After these evaluations, we know we are sending message m to receiver r of class c, with actual parameters $v_1, \ldots, v_n$. The method is evaluated with store σ_n.
- The premise $m \rhd c \mathbin{@} \text{USER\_METHOD}(\_, \langle x_1, \ldots, x_n\rangle, \langle y_1, \ldots, y_k\rangle, e_m, \mathtt{s})$ shows that this send executes a method with formal parameters $x_1, \ldots, x_n$, local variables $y_1, \ldots, y_k$, body e_m, and static superclass `s`.
- The equation for $\hat{\sigma}$ allocates fresh, mutable locations to hold the actual parameters and the local variables of the method. The local variables are initialized to *nil*.
- The equation for ρ' creates a new local environment holding only the instance variables of the receiver. Finally, the last premise shows the evaluation of the body of the method e_m, in the new environment created by binding instance variables, actual parameters, and local variables.

We do not specify `instanceVars` formally. Calling `instanceVars`(r), which is defined in chunk 487a, takes the representation r of an object and returns an environment mapping the names of that object's instance variables (and also `self`) to the locations containing those instance variables.

If a message is sent to SUPER, the method search takes place on c_{super}, the static superclass of the current method, not on the class of the receiver. Otherwise the rule is the same.

$$\frac{\begin{array}{c} m \neq \text{VALUE} \\ \langle \text{SUPER}, \rho, c_{\text{super}}, \xi, \sigma\rangle \Downarrow \langle\langle c, r\rangle, \sigma_0\rangle \\ \langle e_1, \rho, c_{\text{super}}, \xi, \sigma_0\rangle \Downarrow \langle v_1, \sigma_1\rangle \quad \cdots \quad \langle e_n, \rho, c_{\text{super}}, \xi, \sigma_{n-1}\rangle \Downarrow \langle v_n, \sigma_n\rangle \\ m \rhd c_{\text{super}} \mathbin{@} \text{USER\_METHOD}(\_, \langle x_1, \ldots, x_n\rangle, \langle y_1, \ldots, y_k\rangle, e_m, \mathtt{s}) \\ \ell_1, \ldots, \ell_n \notin \operatorname{dom}\sigma_n \qquad \ell'_1, \ldots, \ell'_k \notin \operatorname{dom}\sigma_n \\ \ell_1, \ldots, \ell_n, \ell'_1, \ldots, \ell'_k \text{ all distinct} \\ \hat{\sigma} = \sigma_n\{\ell_1 \mapsto v_1, \ldots \ell_n \mapsto v_n, \ell'_1 \mapsto \mathit{nil}, \ldots \ell'_k \mapsto \mathit{nil}\} \\ \rho' = \mathtt{instanceVars}(r) \\ \langle e_m, \rho'\{x_1 \mapsto \ell_1, \ldots, x_n \mapsto \ell_n, y_1 \mapsto \ell'_1, \ldots, y_k \mapsto \ell'_k\}, \mathtt{s}, \xi, \hat{\sigma}\rangle \Downarrow \langle v, \sigma'\rangle \end{array}}{\langle \text{SEND}(m, \text{SUPER}, e_1, \ldots, e_n), \rho, c_{\text{super}}, \xi, \sigma\rangle \Downarrow \langle v, \sigma'\rangle} \quad \text{(SENDSUPER)}$$

The rules for primitive methods are slightly simpler, since a function is bound directly to the method.

$$\frac{\begin{array}{c} e \neq \text{SUPER} \qquad m \neq \text{VALUE} \\ \langle e, \rho, c_{\text{super}}, \xi, \sigma\rangle \Downarrow \langle\langle c, r\rangle, \sigma_0\rangle \\ \langle e_1, \rho, c_{\text{super}}, \xi, \sigma_0\rangle \Downarrow \langle v_1, \sigma_1\rangle \quad \cdots \quad \langle e_n, \rho, c_{\text{super}}, \xi, \sigma_{n-1}\rangle \Downarrow \langle v_n, \sigma_n\rangle \\ m \rhd c \mathbin{@} \text{PRIM\_METHOD}(\_, f) \\ f(\langle c, r\rangle, \langle v_1, \ldots, v_n\rangle, \sigma_n) = \langle v, \sigma'\rangle \end{array}}{\langle \text{SEND}(m, e, e_1, \ldots, e_n), \rho, c_{\text{super}}, \xi, \sigma\rangle \Downarrow \langle v, \sigma'\rangle} \quad \text{(SENDPRIM)}$$

Because f is an ML function, it might change the store, so in the operational semantics we show σ_n as a parameter and σ' as a result of f.

$$\frac{\begin{array}{c} m \neq \text{VALUE} \\ \langle \text{SUPER}, \rho, c_{\text{super}}, \xi, \sigma\rangle \Downarrow \langle\langle c, r\rangle, \sigma_0\rangle \\ \langle e_1, \rho, c_{\text{super}}, \xi, \sigma_0\rangle \Downarrow \langle v_1, \sigma_1\rangle \quad \cdots \quad \langle e_n, \rho, c_{\text{super}}, \xi, \sigma_{n-1}\rangle \Downarrow \langle v_n, \sigma_n\rangle \\ m \rhd c_{\text{super}} \mathbin{@} \text{PRIM\_METHOD}(\_, f) \\ f(\langle c, r\rangle, \langle v_1, \ldots, v_n\rangle, \sigma_n) = \langle v, \sigma'\rangle \end{array}}{\langle \text{SEND}(m, \text{SUPER}, e_1, \ldots, e_n), \rho, c_{\text{super}}, \xi, \sigma\rangle \Downarrow \langle v, \sigma'\rangle} \quad \text{(SENDPRIMSUPER)}$$

The last rule is for sending VALUE to a block. It is much like the rule for evaluating a user-defined method, except there are no local variables or instance variables to worry about.

$$\frac{\begin{array}{c} \langle e, \rho, c_{\text{super}}, \xi, \sigma\rangle \Downarrow \langle\langle c, \text{CLOSURE}(\langle x_1, \ldots, x_n\rangle, es, \mathtt{s}_c, \rho_c)\rangle, \sigma_0\rangle \\ \langle e_1, \rho, c_{\text{super}}, \xi, \sigma_0\rangle \Downarrow \langle v_1, \sigma_1\rangle \quad \cdots \quad \langle e_n, \rho, c_{\text{super}}, \xi, \sigma_{n-1}\rangle \Downarrow \langle v_n, \sigma_n\rangle \\ \ell_1, \ldots, \ell_n \notin \operatorname{dom}\sigma_n \qquad \ell_1, \ldots, \ell_n \text{ all distinct} \\ \hat{\sigma} = \sigma_n\{\ell_1 \mapsto v_1, \ldots \ell_n \mapsto v_n\} \\ \langle \text{BEGIN}(es), \rho_c\{x_1 \mapsto \ell_1, \ldots, x_n \mapsto \ell_n\}, \mathtt{s}_c, \xi, \hat{\sigma}\rangle \Downarrow \langle v, \sigma'\rangle \end{array}}{\langle \text{SEND}(\text{VALUE}, e, e_1, \ldots, e_n), \rho, c_{\text{super}}, \xi, \sigma\rangle \Downarrow \langle v, \sigma'\rangle} \quad \text{(SENDVALUE)}$$

Sequential execution BEGIN expressions are as in μScheme.

$$\frac{}{\langle \text{BEGIN}(), \rho, c_{\texttt{super}}, \xi, \sigma\rangle \Downarrow \langle \mathit{nil}, \sigma\rangle} \quad \text{(EMPTYBEGIN)}$$

$$\frac{\langle e_1, \rho, c_{\texttt{super}}, \xi, \sigma_0\rangle \Downarrow \langle v_1, \sigma_1\rangle \quad \cdots \quad \langle e_n, \rho, c_{\texttt{super}}, \xi, \sigma_{n-1}\rangle \Downarrow \langle v_n, \sigma_n\rangle}{\langle \text{BEGIN}(e_1, e_2, \ldots, e_n), \rho, c_{\texttt{super}}, \xi, \sigma_0\rangle \Downarrow \langle v_n, \sigma_n\rangle} \quad \text{(BEGIN)}$$

Definitions

The top-level state of a μSmalltalk machine is a global environment ξ and the contents of the store σ. Evaluating a definition d may change both ξ and σ; we write the judgment $\langle d, \xi, \sigma\rangle \to \langle \xi', \sigma'\rangle$.

Global variables As in μScheme, a VAL binding for an existing variable assigns to that variable's location. To evaluate the right-hand side, we need to use the evaluation form $\langle e, \rho, c_{\texttt{super}}, \xi, \sigma\rangle \Downarrow \langle v, \sigma'\rangle$. This form requires additional context, including an environment ρ and a putative superclass. We use the empty environment, and as the superclass, we use the root class `Object`. We use the original definition of `Object` taken from the initial global enviroment ξ_0, not whatever definition of `Object` happens to be current. (In practice, the identity of the superclass is irrelevant. If a message is sent to `super`, the abstract machine tries to look up `self` in the empty environment, and it gets stuck.)

$$\frac{x \in \operatorname{dom}\xi \qquad \xi(x) = \ell \qquad \langle e, \{\}, \xi_0(\texttt{Object}), \sigma, \xi\rangle \Downarrow \langle v, \sigma'\rangle}{\langle \text{VAL}(x, e), \xi, \sigma\rangle \to \langle \xi, \sigma'\{\ell \mapsto v\}\rangle} \quad \text{(DEFINEOLDGLOBAL)}$$

$$\frac{x \notin \operatorname{dom}\xi \qquad \ell \notin \operatorname{dom}\sigma \qquad \langle e, \{\}, \xi_0(\texttt{Object}), \sigma, \xi\rangle \Downarrow \langle v, \sigma'\rangle}{\langle \text{VAL}(x, e), \xi, \sigma\rangle \to \langle \xi\{x \mapsto \ell\}, \sigma'\{\ell \mapsto v\}\rangle} \quad \text{(DEFINENEWGLOBAL)}$$

Top-level expressions A top-level expression is syntactic sugar for a binding to `it`.

$$\frac{\langle \text{VAL}(\texttt{it}, e), \xi, \sigma\rangle \to \langle \xi, \sigma'\rangle}{\langle \text{EXP}(e), \xi, \sigma\rangle \to \langle \xi, \sigma'\rangle} \quad \text{(EVALEXP)}$$

Block definition DEFINE is syntactic sugar for creating a block.

$$\frac{\langle \text{VAL}(f, \text{BLOCK}(\langle x_1, \ldots, x_n\rangle, e)), \xi, \sigma\rangle \to \langle \xi', \sigma'\rangle}{\langle \text{DEFINE}(f, \langle x_1, \ldots, x_n\rangle, e), \xi, \sigma\rangle \to \langle \xi', \sigma'\rangle} \quad \text{(DEFINEBLOCK)}$$

Class definition The processing of a class definition is complex; the interpreter creates an object to represent the class. The details are hidden in the function `newClassObject`, which we don't specify formally, but you can consult the code in chunk 492a.

$$\frac{x \in \operatorname{dom}\xi \qquad \xi(x) = \ell \qquad v = \texttt{newClassObject}(d, \xi, \sigma)}{\langle \textsc{classd}(d), \xi, \sigma\rangle \to \langle \xi, \sigma\{\ell \mapsto v\}\rangle} \quad (\textsc{DefineOldClass})$$

$$\frac{x \notin \operatorname{dom}\xi \qquad \ell \notin \operatorname{dom}\sigma \qquad v = \texttt{newClassObject}(d, \xi, \sigma)}{\langle \textsc{classd}(d), \xi, \sigma\rangle \to \langle \xi\{x \mapsto \ell\}, \sigma\{\ell \mapsto v\}\rangle} \quad (\textsc{DefineNewClass})$$

10.6.3 Structure of the interpreter

In the other languages in this book, values are simple. Not in μSmalltalk. Putting together the built-in objects and values is the most complex part of the interpreter. Some of the complexity comes from primitives: μSmalltalk has almost twice as many primitives as μScheme. But most of the complexity comes from circular dependencies that are built into the specification of the predefined classes. The first circularity involves the Booleans.

1. Value `true` must be an instance of class `Boolean`.
2. Class `Boolean` must be a subclass of class `Object`.
3. Class `Object` has method `notNil`.
4. Method `notNil` must return `true` on class `Object`.

Each of these things depends on all the others, and we must find a way to break the cycle. Value `false` and method `nil` participate in a similar cycle.

The second circularity involves literals.

1. When the evaluator sees an integer literal, it must create an integer value.
2. That integer value must be of class `Integer`.[13]
3. Class `Integer` has to be read from the initial basis.
4. We need the evaluator to read the basis.[14]

Again we have a cycle.

We break each cycle with a strategically placed ref cell. We put an unusable value in each ref cell, bootstrap the interpreter by reading the initial basis, then update the ref cells to hold the proper values. The update operations close the cycles.

[13] Our interpreter supports only `SmallInteger`, but full Smalltalk-80 supports literals that are too large to fit in `SmallInteger`.

[14] This last claim isn't strictly true—we could, if we wanted, avoid `val` bindings in the initial basis, make `eval` a parameter to `evaldef`, and use `evaldef` without `eval` to process only class definitions. But this strategy is too baroque for my taste.

The ref cells and the functions that update them are defined in the chunk ⟨*bootstrapping support* 476c⟩. Primitive methods and built-in objects depend on these ref cells, as does the parser. The evaluator is built on top of everything else, and finally ⟨*initialization* 494c⟩ reads the initial basis, then closes the cycles by calling the functions from ⟨*bootstrapping support* 476c⟩.

The code in the interpreter is organized so that the ⟨*bootstrapping support* 476c⟩ is as early as possible, immediately following the definition of ⟨*abstract syntax and values* (generated automatically)⟩. Afterward come parsing, primitives, and evaluation. The code for ⟨*initialization* 494c⟩ comes almost at the end, just before the execution of the command line. The full structure of the interpreter is as follows:

475a ⟨*usm.sml* 475a⟩≡ (0—1)

```
⟨environments 214⟩
⟨lexical analysis 699a⟩
⟨abstract syntax and values 467a⟩
⟨bootstrapping support 476c⟩
⟨parsing 700b⟩
⟨primitive basics 478b⟩
⟨primitive methods for Object and UndefinedObject 479b⟩
⟨built-in classes Object and UndefinedObject 485a⟩
⟨primitive methods for remaining classes 480f⟩
⟨remaining built-in classes 485d⟩
⟨implementation of use, with Boolean echo 707b⟩
⟨evaluation 486a⟩
⟨initialization 494c⟩
⟨command line 223d⟩
```

The structure of the interpreter has one more subtlety. Before we can define values of the built-in classes, we have to define the classes themselves. And before we can define the built-in classes, we have to define the primitive methods that are used in those classes. But there are primitive methods that depend on `nil`, which is a value of a built-in class! For example, when we create a new array, its contents are initially `nil`. So we structure the primitive methods and the built-in classes in two layers. The first layer includes chunks ⟨*primitive methods for* `Object` *and* `UndefinedObject` 479b⟩ and ⟨*built-in classes* `Object` *and* `UndefinedObject` 485a⟩. This code defines `Object`, which enables us to define `UndefinedObject`, which enables us to define `nilValue` (the internal representation of `nil`). The second layer includes chunks ⟨*primitive methods for remaining classes* 480f⟩ and ⟨*remaining built-in classes* 485d⟩. They define all the other primitive methods and built-in classes, some of which use `nilValue`.

10.6.4 Creating the primitive classes and values

Utilities for manipulating classes

CLASS 468b

Because a class can point to its superclass, the type `class` has to be a recursive type implemented as an ML `datatype`. To get access to information about a class, we have to write a pattern match. When all we want is a class's name or its unique identifier, pattern matching is fairly heavy notation, so we provide two convenience functions.

475b ⟨*abstract syntax and values* 467a⟩+≡ (475a) ◁469 476a▷

```
fun className (CLASS {name, ...}) = name
fun classId   (CLASS {id,   ...}) = id
```

```
className : class -> name
classId   : class -> int
```

We extract a name from a method using another convenience function, `methodName`. Other manipulations of methods include `renameMethod`, which is used when a user class wants to use a primitive method with a name other than the one we built in, and `methods`, which builds an environment suitable for use in a class.

476a ⟨*abstract syntax and values* 467a⟩+≡ (475a) ◁475b 476b▷

```
methodName   : method -> name
methods      : method list -> method env
renameMethod : name * method -> method
```

```
fun methodName (PRIM_METHOD (n, _)) = n
  | methodName (USER_METHOD { name, ... }) = name
fun renameMethod (n, PRIM_METHOD (_, f)) = PRIM_METHOD (n, f)
  | renameMethod _ = raise InternalError "renamed user method"
fun methods l = foldl (fn (m, rho) => bind (methodName m, m, rho)) emptyEnv l
```

To build a class, we keep a private supply of unique identifiers. Identifiers below 10 are reserved for built-in classes not created with `mkClass`; the current implementation uses only id 1, for `Object`, and id 0, for a bootstrapping class. No class ever has a negative identifier.

476b ⟨*abstract syntax and values* 467a⟩+≡ (475a) ◁476a

```
mkClass : name -> class -> name list -> method list -> class
```

```
local
  val n = ref 10 (* new classes start here *)
  fun uid () = !n before n := !n + 1
in
  fun mkClass name super ivars ms =
    CLASS { name = name, super = SOME super, ivars = ivars, methods = methods ms,
            id = uid () }
end
```

Support for bootstrapping the interpreter

Literals An integer literal should be of class `SmallInteger`, which means that before we can evaluate an integer literal, we need to have read the definition of `SmallInteger` from the initial basis. But in order to read the initial basis, we need the evaluator!

We break the circularity by defining a function `mkInteger`. If called before the interpreter is bootstrapped, `mkInteger` fails. If called afterward, `mkInteger` creates a value of class `SmallInteger`. We treat symbols and arrays in similar fashion.

ARRAY 468a
bind 214
CLASS 468b
type class 468b
emptyEnv 214
InternalError 469
NUM 468a
PRIM_METHOD 468c
SYM 468a
USER_METHOD 468c

⟨*bootstrapping support* 476c⟩≡ (475a) 477a▷

```
mkInteger : int         -> value
mkSymbol  : string      -> value
mkArray   : value list -> value
```

```
local
  val intClass    = ref NONE : class option ref
  val symbolClass = ref NONE : class option ref
  val arrayClass  = ref NONE : class option ref
  fun badlit what =
    raise InternalError
        ("(bootstrapping) -- cannot " ^ what ^ " anywhere in initial basis")
in
  fun mkInteger n = (valOf (!intClass), NUM n)
    handle Option => badlit "evaluate integer literal or use array literal"

  fun mkSymbol s = (valOf (!symbolClass), SYM s)
    handle Option => badlit "evaluate symbol literal or use array literal"

  fun mkArray a = (valOf (!arrayClass), ARRAY (Array.fromList a))
    handle Option => badlit "use array literal"
```

Function `valOf` and exception `Option` are part of the initial basis of Standard ML.

Once the initial basis has been read, the initialization code calls `closeLiteralsCycle` to assign the appropriate classes to the ref cells. The environment containing the initial basis is passed as the parameter `xi`.[15]

477a ⟨*bootstrapping support* 476c⟩+≡ (475a) ◁476c 477b▷

```
findInitialClass : string * value ref env -> class
```

```
fun findInitialClass (name, xi) =
      case !(find (name, xi))
        of (_, CLASSREP c) => c
         | _ => raise InternalError (name ^ " is not a class in the initial basis")
fun closeLiteralsCycle xi =
  ( intClass    := SOME (findInitialClass ("SmallInteger", xi))
  ; symbolClass := SOME (findInitialClass ("Symbol",       xi))
  ; arrayClass  := SOME (findInitialClass ("Array",        xi))
  )
end
```

Because it's hard to reason about ref cells, we protect them inside `local`. This protection guarantees that `mkInteger` and `closeLiteralsCycle` are the *only* functions whose behavior can depend on the contents of `intClass`, and similarly for the other functions and ref cells.

Booleans We use the same technique for Booleans, except instead of saving classes, we save values.

477b ⟨*bootstrapping support* 476c⟩+≡ (475a) ◁477a 478a▷

```
mkBoolean : bool -> value
```

```
local
  val trueValue  = ref NONE : value option ref
  val falseValue = ref NONE : value option ref
in
  fun mkBoolean b = valOf (!(if b then trueValue else falseValue))
    handle Option =>
        raise InternalError
            "Bad booleanClass; evaluated boolean expression in initial basis?"
  fun closeBooleansCycle xi =
    ( trueValue  := SOME (!(find ("true",  xi)))
    ; falseValue := SOME (!(find ("false", xi)))
    )
end
```


[15] Xi is the Greek letter ξ, pronounced "ksee."

Blocks We use the technique again for blocks. It is not actually required for blocks, but by declaring `Block` and `Boolean` together, we help clarify the relationship between them, especially the implementations of the `whileTrue:` and `whileFalse:` methods.

478a ⟨*bootstrapping support* 476c⟩+≡ (475a) ◁477b

```
mkBlock : name list * exp list * value ref env * class -> value
local
  val blockClass = ref NONE : class option ref
in
  fun mkBlock c = (valOf (!blockClass), CLOSURE c)
    handle Option =>
        raise InternalError
            "Bad blockClass; evaluated block expression in initial basis?"
  fun closeBlocksCycle xi =
    blockClass := SOME (findInitialClass ("Block", xi))
end
```

10.6.5 Primitives

Utilities for creating primitives

We create most primitives directly from functions written in ML. Here we turn unary and binary functions into primitives, then turn primitives into methods.

478b ⟨*primitive basics* 478b⟩≡ (475a) 479a▷

```
unaryPrim  : (value          -> value) -> primitive
binaryPrim : (value * value -> value) -> primitive
primMethod : name -> primitive -> method
```

```
type primitive = value * value list -> value
fun arityError n (receiver, args) =
  raise RuntimeError ("primitive method expected " ^ Int.toString n ^
                      " arguments; got " ^ Int.toString (length args))
fun unaryPrim  f = (fn (a, [])  => f  a     | ra => arityError 0 ra)
fun binaryPrim f = (fn (a, [b]) => f (a, b) | ra => arityError 1 ra)
fun primMethod name f = PRIM_METHOD (name, f)
```

type class 468b
CLOSURE 468a
findInitialClass 477a
InternalError 469
PRIM_METHOD 468c
RuntimeError 469
type value 468d

A few primitives are more easily created as user methods. For them, we define function `userMethod`. The only dodgy bit is the `superclass` field of the user method. Because this class is used only to define the meaning of messages to `super`, and because none of our predefined user methods sends messages to `super`, we can get away with a bogus class that understands no messages.

Function `exp` is an auxiliary function used to parse a string into abstract syntax.

479a ⟨*primitive basics* 478b⟩+≡ (475a) ◁478b

`internalExp : string -> exp`

```
fun userMethod name formals locals body =
  let val bogusSuper = CLASS { name = "should never be used", super = NONE,
                               ivars = [], methods = [], id = 0 }
  in  USER_METHOD { name = name, formals = formals, locals = locals,
                    body = internalExp body, superclass = bogusSuper }
  end
and internalExp s =
  let val name = "internal expression \"" ^ s ^ "\""
      val parser = exp <?> "expression"
      val input = reader (smalltalkToken, parser) noPrompts (name, streamOfList [s])
      exception BadUserMethodInInterpreter of string (* can't be caught *)
  in  case streamGet input
        of SOME (e, _) => e
         | NONE => raise BadUserMethodInInterpreter s
  end
```

Object primitives

Equality Smalltalk defines equality as object identity: two values are equal if and only if they are the same object. Our primitive implementation compares representations directly, using ML primitives. Here's how we justify the cases:

- ML equality on arrays *is* object identity.
- Because numbers and symbols are immutable in both Smalltalk and ML, we can use ML equality on numbers and symbols, and it appears to the μSmalltalk programmer that we are using object identity.
- The `USER` representation is an environment containing mutable reference cells. ML's `ref` function is also generative, so ML equality on ref cells is object identity. Comparing the representation of two `USER` objects compares their instance-variable environments, which are equal only if they contain the same ref cells, which is possible only if they represent the same μSmalltalk object.
- Blocks, which are represented as closures, always compare unequal, even when compared with themselves.
- Two classes are the same object if and only if they have the same unique identifier.

<?> 663c
ARRAY 468a
CLASS 468b
classId 475b
CLASSREP 468a
CLOSURE 468a
exp 702
noPrompts 668d
NUM 468a
reader 669
smalltalkToken 700a
streamGet 647b
streamOfList 647c
SYM 468a
USER 468a
USER_METHOD 468c

479b ⟨*primitive methods for* `Object` *and* `UndefinedObject` 479b⟩≡ (475a) 480b▷

`eqRep : value * value -> bool`

```
fun eqRep ((cx, x), (cy, y)) =
  classId cx = classId cy andalso
  case (x, y)
    of (ARRAY x,    ARRAY    y) => x = y
     | (NUM   x,    NUM      y) => x = y
     | (SYM   x,    SYM      y) => x = y
     | (USER  x,    USER     y) => x = y
     | (CLOSURE  x, CLOSURE  y) => false
     | (CLASSREP x, CLASSREP y) => classId x = classId y
     | _ => false
```

480a ⟨*primitive methods* :: 480a⟩≡ (492c) 480c▷
```
primMethod "eqObject" (binaryPrim (mkBoolean o eqRep)) ::
```

Printing By default, an object prints as its class name in angle brackets.

480b ⟨*primitive methods for* `Object` *and* `UndefinedObject` 479b⟩+≡ (475a) ◁479b 480d▷
```
fun defaultPrint (self as (c, _)) = ( app print ["<", className c, ">"]; self )
```

480c ⟨*primitive methods* :: 480a⟩+≡ (492c) ◁480a 481c▷
```
primMethod "print" (unaryPrim defaultPrint) ::
```

Class membership For `memberOf`, the class `c` of `self` has to be the same as the class `c'` of the argument. For `kindOf`, it just has to be a subclass.

480d ⟨*primitive methods for* `Object` *and* `UndefinedObject` 479b⟩+≡ (475a) ◁480b 480e▷
```
fun memberOf ((c, _), (_, CLASSREP c')) = mkBoolean (classId c = classId c')
  | memberOf _ = raise RuntimeError "argument of isMemberOf: must be a class"

fun kindOf ((c, _), (_, CLASSREP (CLASS {id=u', ...}))) =
      let fun subclassOfClassU' (CLASS {super, id=u, ... }) =
            u = u' orelse (case super of NONE => false | SOME c => subclassOfClassU' c)
      in  mkBoolean (subclassOfClassU' c)
      end
  | kindOf _ = raise RuntimeError "argument of isKindOf: must be a class"
```

The `error:` primitive raises `RuntimeError`.

480e ⟨*primitive methods for* `Object` *and* `UndefinedObject` 479b⟩+≡ (475a) ◁480d
```
fun error (_, (_, SYM s)) = raise RuntimeError s
  | error (_, (c, _   )) =
      raise RuntimeError ("error: got class " ^ className c ^ "; expected Symbol")
```

Integer primitives

To print integers, we can't use ML's `Int.toString` directly, because `Int.toString` renders a minus sign as `~`. We use `String.map` to convert `~` to `-`.

⟨*primitive methods for remaining classes* 480f⟩≡ (475a) 481a▷
```
fun printInt (self as (_, NUM n)) =
      ( print (String.map (fn #"~" => #"-" | c => c) (Int.toString n))
      ; self
      )
  | printInt _ =
      raise RuntimeError ("cannot print when object inherits from Int")
```


binaryPrim 478b
CLASS 468b
classId 475b
className 475b
CLASSREP 468a
eqRep 479b
mkBoolean 477b
NUM 468a
primMethod 478b
RuntimeError 469
SYM 468a
unaryPrim 478b

A binary integer operation created with `arith` expects as arguments two integers `m` and `n`; it applies an operation `$` to them and uses a creator function `mk` to convert the result to a value. We use `binaryInt` to build arithmetic and comparison.

481a ⟨*primitive methods for remaining classes* 480f⟩+≡ (475a) ◁480f 481b▷

```
binaryInt  : ('a -> value) -> (int * int -> 'a)   -> value * value -> value
arithop    :                  (int * int -> int)  -> primitive
intcompare :                  (int * int -> bool) -> primitive
```

```
fun binaryInt mk $ ((_, NUM n), (_, NUM m)) = mk ($ (n, m))
  | binaryInt _ _  ((_, NUM n), (c, _)) =
      raise RuntimeError ("numeric primitive expected numeric argument, got <"
                          ^ className c ^ ">")
  | binaryInt _ _  ((c, _), _) =
      raise RuntimeError ("numeric primitive method defined on <" ^ className c ^ ">")
fun arithop    $ = binaryPrim (binaryInt mkInteger $)
fun intcompare $ = binaryPrim (binaryInt mkBoolean $)
```

To create a new integer, you must pass an argument that is represented by an integer.

481b ⟨*primitive methods for remaining classes* 480f⟩+≡ (475a) ◁481a 481d▷

```
fun newInteger ((_, CLASSREP c), (_, NUM n)) = (c, NUM n)
  | newInteger _ = raise RuntimeError ("made new integer with non-int or non-class")
```

Here are the primitive operations on small integers.

481c ⟨*primitive methods* :: 480a⟩+≡ (492c) ◁480c 481f▷

```
primMethod "printSmallInteger" (unaryPrim printInt) ::
primMethod "newSmallInteger:"  (binaryPrim newInteger) ::
primMethod "+"   (arithop op +  )  ::
primMethod "-"   (arithop op -  )  ::
primMethod "*"   (arithop op *  )  ::
primMethod "div" (arithop op div)  ::
primMethod "<"   (intcompare op <) ::
primMethod ">"   (intcompare op >) ::
```

In chunk 461e, these primitive methods are used to define class `SmallInteger`.

Symbol primitives

A symbol prints as its name, with no leading `#`.

481d ⟨*primitive methods for remaining classes* 480f⟩+≡ (475a) ◁481b 481e▷

```
fun printSymbol (self as (_, SYM s)) = (print s; self)
  | printSymbol _ = raise RuntimeError "cannot print when object inherits from Symbo
```

To create a new symbol, you must pass an argument that is represented by a symbol.

481e ⟨*primitive methods for remaining classes* 480f⟩+≡ (475a) ◁481d 482b▷

```
fun newSymbol ((_, CLASSREP c), (_, SYM s)) = (c, SYM s)
  | newSymbol _ = raise RuntimeError ("made new symbol with non-symbol or non-class")
```

481f ⟨*primitive methods* :: 480a⟩+≡ (492c) ◁481c 482e▷

```
primMethod "printSymbol" (unaryPrim  printSymbol) ::
primMethod "newSymbol"   (binaryPrim newSymbol  ) ::
```

There is no need to create `Symbol` internally, so we put it in the initial basis.

482a ⟨*additions to the μSmalltalk initial basis* 446⟩+≡ ◁448a 495d▷

```
(class Symbol Object
    () ; internal representation
    (class-method new () (error: self #can't-send-new-to-Symbol))
    (class-method new:   primitive newSymbol)
    (method       print  primitive printSymbol)
)
```

Array primitives

The primitive operations on arrays are creation, subscript, update, and size.

A new array contains all `nil`.

482b ⟨*primitive methods for remaining classes* 480f⟩+≡ (475a) ◁481e 482c▷

```
fun newArray ((_, CLASSREP c), (_, NUM n)) = (c, ARRAY (Array.array (n, nilValue)))
  | newArray _ = raise RuntimeError "Array new sent to non-class or got non-integer"
```

To create primitives that expect `self` to be an array, we define `arrayPrimitive`.

482c ⟨*primitive methods for remaining classes* 480f⟩+≡ (475a) ◁482b 482d▷

```
arrayPrimitive : (value array * value list -> value) -> primitive
```

```
fun arrayPrimitive f ((_, ARRAY a), l) = f (a, l)
  | arrayPrimitive f _ = raise RuntimeError "Array primitive used on non-array"

fun arraySize (a, []) = mkInteger (Array.length a)
  | arraySize ra      = arityError 0 ra
```

When defining array primitives for `at:` and `at:put:`, we adjust for differing indexing conventions: in Smalltalk, arrays are indexed from 1, but in ML, arrays are indexed from 0.

482d ⟨*primitive methods for remaining classes* 480f⟩+≡ (475a) ◁482c 483a▷

```
fun arrayAt (a, [(_, NUM n)]) = Array.sub (a, n - 1)  (* convert to 0-indexed *)
  | arrayAt (_, [_])  = raise RuntimeError "Non-integer used as array subscript"
  | arrayAt ra        = arityError 1 ra

fun arrayAtPut (a, [(_, NUM n), x]) = (Array.update (a, n-1, x); x)
  | arrayAtPut (_, [_, _]) = raise RuntimeError "Non-integer used as array subscript"
  | arrayAtPut ra          = arityError 2 ra
```

Here are all the primitive array methods.

⟨*primitive methods* :: 480a⟩+≡ (492c) ◁481f 482f▷

```
primMethod "arrayNew:"    (binaryPrim     newArray)   ::
primMethod "arraySize"    (arrayPrimitive arraySize)  ::
primMethod "arrayAt:"     (arrayPrimitive arrayAt)    ::
primMethod "arrayAt:Put:" (arrayPrimitive arrayAtPut) ::
```

In chunk 458b, these primitive methods are used to define class `Array`.

Block primitives

Actually, `value` is not a primitive; it is built into `eval`. We create a primitive method called `value` anyway, which we use in the definition of class `Block` (page 448), so if there's a bug in the interpreter, we get an informative error message.

482f ⟨*primitive methods* :: 480a⟩+≡ (492c) ◁482e 519b▷

```
primMethod "value" (fn _ => raise InternalError "hit primitive method 'value'") ::
```

arityError 478b
ARRAY 468a
binaryPrim 478b
CLASSREP 468a
InternalError 469
mkInteger 476c
nilValue 485c
NUM 468a
Object 8
primMethod 478b
RuntimeError 469

Class primitives

The class primitives take both the metaclass and the class as arguments.

483a ⟨*primitive methods for remaining classes* 480f⟩+≡ (475a) ◁482d 483b▷

```
classPrimitive : (class * class -> value) -> primitive
fun classPrimitive f =
  unaryPrim (fn (meta, CLASSREP c) => f (meta, c)
              | _ => raise RuntimeError "class primitive sent to non-class")
```

Object creation The most important primitive defined on classes is `new`. To create a new object, we allocate fresh instance variables, each containing `nilValue`. Given the variables, we can allocate the object, and finally we assign `self` to point to the object itself.

483b ⟨*primitive methods for remaining classes* 480f⟩+≡ (475a) ◁483a 484▷

```
mkIvars       : class -> value ref env
newUserObject : class * class -> value
local
  fun mkIvars (CLASS { ivars, super, ... }) =
    let val supervars = case super of NONE => emptyEnv | SOME c => mkIvars c
    in  foldl (fn (n, rho) => bind (n, ref nilValue, rho)) supervars ivars
    end
in
  fun newUserObject (_, c) =
        let val ivars = mkIvars c
            val self = (c, USER ivars)
        in  (find ("self", ivars) := self; self)
        end
end
```

Showing protocols The `showProtocol` function helps implement the `protocol` and `localProtocol` primitives, which are defined on class `Class`. Its implementation is not very interesting. Function `insert` helps implement an insertion sort, which we use to present methods in alphabetical order.

484 ⟨*primitive methods for remaining classes* 480f⟩+≡ (475a) ◁483b 519a▷

```
⟨definition of separate 218d⟩
local
  fun showProtocol doSuper kind c =
    let fun member x l = List.exists (fn x' : string => x' = x) l
        fun insert (x, []) = [x]
          | insert (x, (h::t)) =
              case compare x h
                of LESS    => x :: h :: t
                 | EQUAL   => x :: t (* replace *)
                 | GREATER => h :: insert (x, t)
        and compare (name, _) (name', _) = String.compare (name, name')
        fun methods (CLASS { super, methods = ms, name, ... }) =
              if not doSuper orelse (kind = "class-method" andalso name = "Class") then
                foldl insert [] ms
              else
                foldl insert (case super of NONE => [] | SOME c => methods c) ms
        fun show (name, PRIM_METHOD _) =
              app print ["(", kind, " ", name, " primitive ...)\n"]
          | show (name, USER_METHOD { formals, ... }) =
              app print ["(", kind, " ", name,
                         " (", spaceSep formals, ") ...)\n"]
    in  app show (methods c)
    end
in
  fun protocols all (meta, c) =
    ( showProtocol all "class-method" meta
    ; showProtocol all "method" c
    ; (meta, CLASSREP c)
    )
end
```

10.6.6 The built-in classes

CLASS 468b
CLASSREP 468a
PRIM_METHOD 468c
spaceSep 218d
USER_METHOD 468c

Almost all classes are defined in the initial basis; only three need to be built into the interpreter:

- `Object` must be built in because it has no superclass, and because everything else must inherit from it.
- `UndefinedObject` must be built in because it is `nil`'s class, and `nil` must be built in because both the evaluator and a number of primitives need `nil`.
- `Class` must be built in because it is `Object`'s metaclass. That is, the μSmalltalk value `Object`, which represents a class, is an instance of `Class`.[16]

Every other class in the initial basis is defined by a class definition written in μSmalltalk itself, possibly using primitive methods.

[16] Actually it's an instance of `Object`'s metaclass, which inherits from `Class`.

The distinguished root class `Object` Class `Object` is the ultimate superclass. By putting `self` in its representation, we ensure that every object has an instance variable called `self`. We also make sure that every object responds to messages in the `Object` protocol described in Figure 10.4 on page 412.

485a ⟨*built-in classes* `Object` *and* `UndefinedObject` 485a⟩≡ (475a) 485b▷

```
val objectClass =
  CLASS { name = "Object", super = NONE, ivars = ["self"], id = 1
        , methods = methods
            [ primMethod "print"   (unaryPrim defaultPrint)
            , userMethod "println" [] [] "(begin (print self) (print newline) self)"
            , primMethod "isNil"   (unaryPrim (fn _ => mkBoolean false))
            , primMethod "notNil"  (unaryPrim (fn _ => mkBoolean true))
            , primMethod "error:"  (binaryPrim error)
            , primMethod "="       (binaryPrim (mkBoolean o eqRep))
            , userMethod "!=" ["x"] [] "(not (= self x))"
            , primMethod "isKindOf:"    (binaryPrim kindOf)
            , primMethod "isMemberOf:"  (binaryPrim memberOf)
            , primMethod "subclassResponsibility"
              (unaryPrim
                 (fn _ => raise RuntimeError
                  "subclass failed to implement a method that was its responsibility"))
            ]
        }
```

The undefined object Class `UndefinedObject`, whose sole instance is `nil`, redefines `isNil`, `notNil`, and `print`.

485b ⟨*built-in classes* `Object` *and* `UndefinedObject` 485a⟩+≡ (475a) ◁485a 485c▷

```
val nilClass =
  mkClass "UndefinedObject" objectClass []
    [ primMethod "isNil"  (unaryPrim (fn _ => mkBoolean true))
    , primMethod "notNil" (unaryPrim (fn _ => mkBoolean false))
    , primMethod "print"  (unaryPrim (fn x => (print "nil"; x)))
    ]
```

To create the `nil` value, we have to bind `self`; otherwise `println` won't work on `nil`.

485c ⟨*built-in classes* `Object` *and* `UndefinedObject` 485a⟩+≡ (475a) ◁485b

```
val nilValue =
  let val nilCell  = ref (nilClass, USER []) : value ref
      val nilValue = (nilClass, USER (bind ("self", nilCell, emptyEnv)))
      val _        = nilCell := nilValue
  in  nilValue
  end
```

Class `Class` Class `Class` is in the interpreter so that metaclasses can inherit from it. As explained in Figure 10.5 on page 413, the methods that are defined on class `Class`, and therefore defined on all class objects, are `new`, `protocol`, and `localProtocol`.

485d ⟨*remaining built-in classes* 485d⟩≡ (475a)

```
val classClass =
  mkClass "Class" objectClass []
    [ primMethod "new"           (classPrimitive newUserObject)
    , primMethod "protocol"      (classPrimitive (protocols true))
    , primMethod "localProtocol" (classPrimitive (protocols false))
    ]
```

10.6.7 Evaluation

Except at top level, an expression is evaluated only as part of a message send. The context of the message send determines each name's meaning: global, instance variable, formal parameter, or local variable. In μScheme, we could bind all those names in a single environment. But in μSmalltalk, as discussed in Section 10.6.2, it is permissible for a method to refer to a global variable bound only after the method is defined. We therefore split the world into two environments: `xi`, which binds the global variables, and `rho`, which binds everything else. Environment `xi` gives the meanings of global variables, and it is threaded through the interpreter. Environment `rho` gives the meanings of variables that are private to objects and methods. When we create a closure, we capture only `rho`, not `xi`.

We need one more piece of context. To evaluate a message sent to `super`, we have to know the *static* superclass of the method definition within which we are currently evaluating. This class need *not* be the same as the superclass of the object that received the message; see Exercise 38.

The evaluator therefore takes four arguments: an expression to be evaluated; a local environment, which binds instance variables, formal parameters, and local variables; a class used to send message to `super`; and the global environment. These four arguments correspond to the e, ρ, $c_{\texttt{super}}$, and ξ used in the evaluation judgment $\langle e, \rho, c_{\texttt{super}}, \xi, \sigma\rangle \Downarrow \langle v, \sigma'\rangle$. As usual, the states σ and σ' represent states of the underlying ML interpreter, and they are not passed explicitly.

486a ⟨*evaluation* 486a⟩≡ (475a) 492a▷

```
eval: exp * value ref env * class * value ref env -> value
ev  : exp -> value
```

```
⟨tracing 705b⟩
fun eval (e, rho, superclass, xi) =
  let ⟨local helper functions of eval 486b⟩
      ⟨function ev, the evaluator proper 488a⟩
  in  ev e
  end
```

Because the rules for finding and evaluating methods are relatively complex, we define several helper functions that are private to `eval`. We then use those functions to define `ev`, which evaluates a single expression in the context of the current `rho`, `superclass`, and `xi`.

The first function, `findMethod`, implements method search. If $m \triangleright c @ imp$, then calling `findMethod (`m`, `c`)` returns imp. Given m and c, if there is no imp such that $m \triangleright c @ imp$, then calling `findMethod (`m`, `c`)` raises the exception `NotFound` m.

CLASS 468b
className 475b
ev 488a
find 214
NotFound 214
RuntimeError 469

⟨*local helper functions of* eval 486b⟩≡ (486a) 487a▷

```
findMethod : name * class -> method
fm         : class        -> method
```

```
fun findMethod (name, class) =
  let fun fm (CLASS { methods, super, ...}) =
        find (name, methods)
        handle NotFound m =>
          case super
            of SOME c => fm c
             | NONE   => raise RuntimeError
                           (className class ^ " does not understand message " ^ m)
  in  fm class
  end
```

To evaluate a primitive method, we apply the method's function. To evaluate a user-defined method, we build a new environment. Function `evalMethod` implements the second part of the SENDUSER rule:

$$\frac{\begin{array}{c} c \neq \text{SUPER} \qquad m \neq \text{VALUE} \\ \langle e, \rho, c_{\text{super}}, \xi, \sigma\rangle \Downarrow \langle\langle c, r\rangle, \sigma_0\rangle \\ \langle e_1, \rho, c_{\text{super}}, \xi, \sigma_0\rangle \Downarrow \langle v_1, \sigma_1\rangle \quad \cdots \quad \langle e_n, \rho, c_{\text{super}}, \xi, \sigma_{n-1}\rangle \Downarrow \langle v_n, \sigma_n\rangle \\ m \rhd c \mathbin{@} \text{USER\_METHOD}(\_, \langle x_1, \ldots, x_n\rangle, \langle y_1, \ldots, y_k\rangle, e_m, \mathtt{s}) \\ \ell_1, \ldots, \ell_n \notin \operatorname{dom}\sigma_n \qquad \ell'_1, \ldots, \ell'_k \notin \operatorname{dom}\sigma_n \\ \ell_1, \ldots, \ell_n, \ell'_1, \ldots, \ell'_k \text{ all distinct} \\ \hat{\sigma} = \sigma_n\{\ell_1 \mapsto v_1, \ldots \ell_n \mapsto v_n, \ell'_1 \mapsto \mathit{nil}, \ldots \ell'_k \mapsto \mathit{nil}\} \\ \rho' = \mathtt{instanceVars}(r) \\ \langle e_m, \rho'\{x_1 \mapsto \ell_1, \ldots, x_n \mapsto \ell_n, y_1 \mapsto \ell'_1, \ldots, y_k \mapsto \ell'_k\}, \mathtt{s}, \xi, \hat{\sigma}\rangle \Downarrow \langle v, \sigma'\rangle \end{array}}{\langle \text{SEND}(m, e, e_1, \ldots, e_n), \rho, c_{\text{super}}, \xi, \sigma\rangle \Downarrow \langle v, \sigma'\rangle} \qquad \text{(SENDUSER)}$$

We compute ρ' as `rho'` using `instanceVars`. We compute $\rho'\{x_1 \mapsto \ell_1, \ldots, x_n \mapsto \ell_n\}$ as `rho_x` and $\rho'\{x_1 \mapsto \ell_1, \ldots, x_n \mapsto \ell_n, y_1 \mapsto \ell'_1, \ldots, y_k \mapsto \ell'_k\}$ as `rho_y`. The $x_1, \ldots, x_n$ are the list `formals`, the $y_1, \ldots, y_n$ are the list `locals`, the $v_1, \ldots, v_n$ are the list `actuals`, and e_m is `body`.

487a ⟨*local helper functions of* `eval` 486b⟩+≡ (486a) ◁486b 487b▷

```
evalMethod   : method * value * value list -> value
instanceVars : value -> value ref env
```

```
fun evalMethod (PRIM_METHOD (name, f), receiver, actuals) = f (receiver, actuals)
  | evalMethod (USER_METHOD { name, superclass, formals, locals, body },
                receiver, actuals) =
      let val rho'  = instanceVars receiver
          val rho_x = bindList (formals, map ref actuals, rho')
          val rho_y = bindList (locals, map (fn _ => ref nilValue) locals, rho_x)
      in  eval (body, rho_y, superclass, xi)
      end
        handle BindListLength =>
            raise RuntimeError
                ("Wrong number of arguments to method " ^ name ^ "; expected (" ^
                 spaceSep (name :: "self" :: formals) ^ ")")
and instanceVars (_, USER rep) = rep
  | instanceVars self = bind ("self", ref self, emptyEnv)
```

If `receiver` is a `USER` object, `self` is already part of its `rep`. For every other kind of object, `instanceVars` creates an environment that binds only `self`.

Function `evalMethod` handles every primitive method except `value`. To send the `value` message to a block, we need to make a recursive call to `eval`. Encapsulating a recursive call to `eval` in a `PRIM_METHOD` callable from `evalMethod` would be difficult; it's much easier to build that ability into a helper function.

487b ⟨*local helper functions of* `eval` 486b⟩+≡ (486a) ◁487a

```
applyClosure : (name list * exp list * value ref env * class) * value list -> value
```

```
fun applyClosure ((formals, body, rho, superclass), actuals) =
  eval (BEGIN body, bindList (formals, map ref actuals, rho), superclass, xi)
  handle BindListLength =>
      raise RuntimeError ("Wrong number of arguments to block; expected " ^
                          "(value <block>" ^ spaceSep formals ^ ")")
```

With these helper functions in place, we can write the evaluator. A `VALUE` node stands for itself.

$$\frac{}{\langle \text{VALUE}(v), \rho, c_{\text{super}}, \xi, \sigma\rangle \Downarrow \langle v, \sigma\rangle} \quad \text{(VALUE)}$$

488a ⟨*function* `ev`, *the evaluator proper* 488a⟩≡ (486a) 488b▷

```
fun ev (VALUE v) = v
```

`ev : exp -> value`

When we see a `LITERAL` node, we call `mkInteger` or `mkSymbol` to build the literal. It is unsafe to call these functions until we have read the initial basis and bootstrapped the interpreter; integer or symbol literals in the initial basis had better appear only inside method definitions. Evaluating such a literal calls `mkInteger` or `mkSymbol`, and if you revisit chunks 476c and 477a, you will see that it is safe to call `mkInteger` only after the interpreter is fully initialized.

$$\frac{}{\langle \text{LITERAL}(\text{NUM}(n)), \rho, c_{\text{super}}, \xi, \sigma\rangle \Downarrow \langle \{\!|\sigma(\xi(\texttt{SmallInteger})), \text{NUM}(n)|\!\}, \sigma\rangle} \quad \text{(LITERALNUMBER)}$$

$$\frac{}{\langle \text{LITERAL}(\text{SYM}(s)), \rho, c_{\text{super}}, \xi, \sigma\rangle \Downarrow \langle \{\!|\sigma(\xi(\texttt{Symbol})), \text{SYM}(s)|\!\}, \sigma\rangle} \quad \text{(LITERALSYMBOL)}$$

488b ⟨*function* `ev`, *the evaluator proper* 488a⟩+≡ (486a) ◁488a 488c▷

```
| ev (LITERAL c) =
    (case c of NUM n => mkInteger n
             | SYM n => mkSymbol n
             | _ => raise InternalError "unexpected literal")
```

The cases for `VAR` and `SET` are as we would expect, given that we have both local and global environments, just as in Impcore.

$$\frac{x \in \operatorname{dom}\rho}{\langle \text{VAR}(x), \rho, c_{\text{super}}, \xi, \sigma\rangle \Downarrow \langle \sigma(\rho(x)), \sigma\rangle} \quad \text{(VAR)}$$

$$\frac{x \notin \operatorname{dom}\rho \qquad x \in \operatorname{dom}\xi}{\langle \text{VAR}(x), \rho, c_{\text{super}}, \xi, \sigma\rangle \Downarrow \langle \sigma(\xi(x)), \sigma\rangle} \quad \text{(GLOBALVAR)}$$

$$\frac{x \in \operatorname{dom}\rho \qquad \rho(x) = \ell \qquad \langle e, \rho, c_{\text{super}}, \xi, \sigma\rangle \Downarrow \langle v, \sigma'\rangle}{\langle \text{SET}(x, e), \rho, c_{\text{super}}, \xi, \sigma\rangle \Downarrow \langle v, \sigma'\{\ell \mapsto v\}\rangle} \quad \text{(ASSIGN)}$$

$$\frac{x \notin \operatorname{dom}\rho \qquad x \in \operatorname{dom}\xi \qquad \xi(x) = \ell \qquad \langle e, \rho, c_{\text{super}}, \xi, \sigma\rangle \Downarrow \langle v, \sigma'\rangle}{\langle \text{SET}(x, e), \rho, c_{\text{super}}, \xi, \sigma\rangle \Downarrow \langle v, \sigma'\{\ell \mapsto v\}\rangle} \quad \text{(ASSIGNGLOBAL)}$$

⟨*function* `ev`, *the evaluator proper* 488a⟩+≡ (486a) ◁488b 488d▷

```
| ev (VAR v) = !(find (v, rho) handle NotFound _ => find (v, xi))
| ev (SET (n, e)) =
    let val v = ev e
        val cell = find (n, rho) handle NotFound _ => find (n, xi)
    in  cell := v; v
    end
```

`SUPER`, when used as an expression, acts just as `self` does.

$$\frac{\langle \text{VAR}(\texttt{self}), \rho, c_{\text{super}}, \xi, \sigma\rangle \Downarrow \langle v, \sigma\rangle}{\langle \text{SUPER}, \rho, c_{\text{super}}, \xi, \sigma\rangle \Downarrow \langle v, \sigma\rangle} \quad \text{(SUPER)}$$

488d ⟨*function* `ev`, *the evaluator proper* 488a⟩+≡ (486a) ◁488c 489a▷

```
| ev (SUPER) = ev (VAR "self")
```

Evaluation of `BEGIN` is as in μScheme.

$$\frac{}{\langle \text{BEGIN}(), \rho, c_{\text{super}}, \xi, \sigma\rangle \Downarrow \langle \mathit{nil}, \sigma\rangle} \quad \text{(EMPTYBEGIN)}$$

$$\frac{\langle e_1, \rho, c_{\text{super}}, \xi, \sigma_0\rangle \Downarrow \langle v_1, \sigma_1\rangle \quad \cdots \quad \langle e_n, \rho, c_{\text{super}}, \xi, \sigma_{n-1}\rangle \Downarrow \langle v_n, \sigma_n\rangle}{\langle \text{BEGIN}(e_1, e_2, \ldots, e_n), \rho, c_{\text{super}}, \xi, \sigma_0\rangle \Downarrow \langle v_n, \sigma_n\rangle} \quad \text{(BEGIN)}$$

489a ⟨*function* `ev`, *the evaluator proper* 488a⟩+≡ (486a) ◁488d 489b▷

```
| ev (BEGIN es) =
    let fun b (e::es, lastval) = b (es, ev e)
          | b (   [], lastval) = lastval
    in  b (es, nilValue)
    end
```

Evaluating a block means capturing the local environment and superclass in a closure.

$$\frac{}{\langle \text{BLOCK}(\langle x_1, \ldots, x_n\rangle, es), \rho, c_{\text{super}}, \xi, \sigma\rangle \Downarrow \langle \{\!|\sigma(\xi(\texttt{Block})), \text{CLOSURE}(\langle x_1, \ldots, x_n\rangle, es, \rho, c_{\text{super}})|\!\}, \sigma\rangle} \quad \text{(MKCLOSURE)}$$

489b ⟨*function* `ev`, *the evaluator proper* 488a⟩+≡ (486a) ◁489a 490▷

```
| ev (BLOCK (formals, body)) = mkBlock (formals, body, rho, superclass)
```

The last case is for message send. The code here implements parts of three rules: SENDUSER, SENDSUPER, and SENDVALUE. In particular, the code here implements the first parts of the rules, in which the receiver and arguments are evaluated and a method is found. The second parts of the SENDUSER and SENDSUPER rules are identical, and they are implemented by function `evalMethod` in chunk 487a.

First we evaluate the receiver and the arguments. Message send dispatches on the receiver, which is used to find the method that defines `message`, *except* when the message is sent to `super`, in which case we use the `superclass` of the currently running method. There is one case in which we do not use dynamic dispatch (`findMethod`); sending the `value` message to a block is built directly into the interpreter. Instead of calling `findMethod`, we call the evaluator recursively through `applyClosure`. This trick makes it impossible for a subclass of `Block` to redefine the `value` method—but there is nothing useful to inherit from `Block`, so creating a subclass would be a pointless exercise.

$$\frac{\begin{array}{c} e \neq \text{SUPER} \qquad m \neq \text{VALUE} \\ \langle e, \rho, c_{\texttt{super}}, \xi, \sigma\rangle \Downarrow \langle\langle c, r\rangle, \sigma_0\rangle \\ \langle e_1, \rho, c_{\texttt{super}}, \xi, \sigma_0\rangle \Downarrow \langle v_1, \sigma_1\rangle \quad \cdots \quad \langle e_n, \rho, c_{\texttt{super}}, \xi, \sigma_{n-1}\rangle \Downarrow \langle v_n, \sigma_n\rangle \\ m \rhd c \mathbin{@} \text{USER\_METHOD}(\_, \langle x_1, \ldots, x_n\rangle, \langle y_1, \ldots, y_k\rangle, e_m, \mathtt{s}) \\ \ell_1, \ldots, \ell_n \notin \operatorname{dom}\sigma_n \qquad \ell'_1, \ldots, \ell'_k \notin \operatorname{dom}\sigma_n \\ \ell_1, \ldots, \ell_n, \ell'_1, \ldots, \ell'_k \text{ all distinct} \\ \hat{\sigma} = \sigma_n\{\ell_1 \mapsto v_1, \ldots \ell_n \mapsto v_n, \ell'_1 \mapsto \mathit{nil}, \ldots \ell'_k \mapsto \mathit{nil}\} \\ \rho' = \texttt{instanceVars}(r) \\ \langle e_m, \rho'\{x_1 \mapsto \ell_1, \ldots, x_n \mapsto \ell_n, y_1 \mapsto \ell'_1, \ldots, y_k \mapsto \ell'_k\}, \mathtt{s}, \xi, \hat{\sigma}\rangle \Downarrow \langle v, \sigma'\rangle \end{array}}{\langle \text{SEND}(m, e, e_1, \ldots, e_n), \rho, c_{\texttt{super}}, \xi, \sigma\rangle \Downarrow \langle v, \sigma'\rangle} \quad \text{(SendUser)}$$

490 ⟨*function* ev, *the evaluator proper* 488a⟩+≡ (486a) ◁489b

```
| ev (SEND (srcloc, message, receiver, args)) =
    let val obj as (class, rep) = ev receiver
        val args = map ev args
        val dispatchingClass = case receiver of SUPER => superclass | _ => class
        ⟨definitions of message-tracing procedures 491⟩
        fun performSend () =
          case (message, rep)
            of ("value", CLOSURE clo) => applyClosure (clo, args)
             | _ => evalMethod (findMethod (message, dispatchingClass), obj, args)

    in  ( traceSend srcloc
        ; traceAnswer srcloc (performSend ())
        )
    end
```


The implementation of message send is further complicated because we have built two trace facilities into the interpreter: one traces every send and answer, and the other prints a stack trace in case of a run-time error. Both facilities are driven by `traceSend` and `traceAnswer`, which are defined below.

We trace every send and answer only if the μSmalltalk variable `&trace` is set to a non-zero number. At every trace, we also decrement `&trace`. The code that builds the tracing output is protected by `fn () => ...`; it is executed only if tracing is turned on.

491 ⟨*definitions of message-tracing procedures* 491⟩≡ (490)

```
fun traceSend (file, line) =
  traceIndent (message, (file, line)) xi (fn () =>
     [file, ", line ", Int.toString line, ": ",
      "Sending message (", spaceSep (message :: map valueString args), ")",
      " to object of class ", className dispatchingClass])
fun traceAnswer (file, line) answer =
  ( outdentTrace xi (fn () =>
       [file, ", line ", Int.toString line, ": ",
        "(", spaceSep (message :: map valueString (obj :: args)), ")",
        " = ", valueString answer])
  ; answer
  )
```

Functions `traceIndent`, `outdentTrace`, and `valueString` are defined in ⟨*tracing* 705b⟩. Chunk ⟨*tracing* 705b⟩ also defines function `showStackTrace`, which is called from chunk 494a to show the stack of active message sends after a run-time error.

10.6.8 Evaluating definitions; the read-eval-print loop

Most definitions are evaluated more or less as in other interpreters, but class definitions require a lot of special-purpose code, most of which is in the function `newClassObject`. This function takes the abstract syntax of a class definition and creates a class object. It also creates a metaclass, on which it defines the class methods.

- Value `class` is the new class being declared, on which the instance methods are defined; `metaclass` is the new metaclass, on which the class methods are defined.
- Value `super` is the superclass from which the new class inherits; `superMeta` is its metaclass. Class `super` is bound into user-defined instance methods, and class `superMeta` is bound into user-defined class methods, to guarantee that messages sent to `SUPER` from within these methods arrive at the proper destination.
- Function `addMethodDefn` processes each method definition, adding it either to the list of class methods or to the list of instance methods for the new class. We use `foldr` to accumulate these lists and place them in `imethods` and `cmethods`.

492a ⟨*evaluation* 486a⟩+≡ (475a) ◁486a 492b▷

⟨*definition of function* primitiveMethod 492c⟩ `primitiveMethod : name -> method`

```
fun newClassObject {name, super, ivars, methods = ms} xi =
  let val (superMeta, super) = findClass (super, xi)
        handle NotFound s => raise RuntimeError ("Superclass " ^ s ^ " not found")
      fun method (kind, name, PRIM_IMPL prim) =
            renameMethod (name, primitiveMethod prim)
        | method (kind, name, USER_IMPL (formals, ls, body)) =
            USER_METHOD { name = name, formals = formals, body = body, locals = ls
                        , superclass = case kind of IMETHOD => super
                                                  | CMETHOD => superMeta
                        }
      fun addMethodDefn (m as (CMETHOD, _, _), (c's, i's)) = (method m :: c's, i's)
        | addMethodDefn (m as (IMETHOD, _, _), (c's, i's)) = (c's, method m :: i's)
      val (cmethods, imethods) = foldr addMethodDefn ([], []) ms
      val metaclassName = "class " ^ name
      val class     = mkClass name          super      ivars imethods
      val metaclass = mkClass metaclassName superMeta  []    cmethods
  in  (metaclass, CLASSREP class)
  end
```

CLASSREP 468a
CMETHOD 467b
find 214
IMETHOD 467b
methods 476a
mkClass 476b
NotFound 214
PRIM_IMPL 467b
renameMethod 476a
RuntimeError 469
USER_IMPL 467b
USER_METHOD 468c
valueString 707a
492D

The object named as a superclass must in fact represent a class, so its representation must be `CLASSREP c`, where `c` is the class it represents. That object is an instance of its metaclass. Function `findClass` returns the metaclass and the class.

⟨*evaluation* 486a⟩+≡ (475a) ◁492a 493a▷

`findClass : name * value ref env -> class * class`

```
and findClass (supername, xi) =
  case !(find (supername, xi))
    of (meta, CLASSREP c) => (meta, c)
     | v => raise RuntimeError ("object " ^ supername ^ " = " ^ valueString v ^
                                " is not a class")
```

To find a primitive method by name, we make the list of primitive methods into an environment, then look up the name in that environment.

492c ⟨*definition of function* primitiveMethod 492c⟩≡ (492a)

```
val primitiveMethods = methods (⟨primitive methods :: 480a⟩ nil)
fun primitiveMethod name =
  find (name, primitiveMethods)
  handle NotFound n => raise RuntimeError ("There is no primitive method named " ^ n)
```

Evaluating definitions

Evaluating a definition computes a new global environment. In a `VAL` binding, the right-hand side is evaluated as if in the context of a message sent to a value of class `Object`. As usual, a top-level expression is syntactic sugar for a binding to `it`. `DEFINE` is syntactic sugar for a definition of block. Evaluating a class definition binds a new class object, as described above.

493a ⟨*evaluation* 486a⟩+≡ (475a) ◁492b 493b▷

```
evaldef : def * bool * value ref env -> value ref env
```

```
fun evaldef (d, echo, xi)  =
  case d
    of VAL (name, e) => doVal echo (name, eval (e, emptyEnv, objectClass, xi), xi)
     | EXP e                     => evaldef (VAL ("it", e),                    echo, xi)
     | DEFINE (name, args, body) => evaldef (VAL (name, BLOCK (args, [body])), echo, xi)
     | CLASSD (d as {name, ...}) => doVal echo (name, newClassObject d xi, xi)
     | USE filename              => use readEvalPrint filename xi
```

Every binding is added by `doVal`, which is optimized so that if a previous binding exists, we replace the binding and do not create a new global environment. This optimization is safe only because no operation in μSmalltalk makes a copy of the global environment.

493b ⟨*evaluation* 486a⟩+≡ (475a) ◁493a 493c▷

```
and doVal echo (name, value, xi)  =
      ( (find (name, xi) := value; xi)
        handle NotFound _ => bind (name, ref value, xi)
      )
      before printValue echo xi value
```

Unlike our other interpreters, our μSmalltalk interpreter doesn't need special-case code to print top-level expressions differently from `val` bindings. In μSmalltalk, *every* value is printed by using its `print` method, which you can redefine. Because a print method writes directly to standard output, we have to define `echo` to be a Boolean, not a function as in other interpreters.

493c ⟨*evaluation* 486a⟩+≡ (475a) ◁493b 494a▷

```
and printValue echo xi v =
  if echo then
    ( eval (SEND (nullsrc, "print", VALUE v, []), emptyEnv, objectClass, xi)
    ; print "\n"
    )
  else
    ()
```

The source location `nullsrc` identifies the `SEND` as something generated internally, rather than read from a file or a list of strings.

The read-eval-print loop

μSmalltalk's read-eval-print loop differs from others in two ways:

- If an error occurs, we call `showStackTrace` and `resetTrace` before continuing.
- Before each top-level evaluation, we call `closeLiteralsCycle`. This call updates the definitions (if any) of `SmallInteger`, `Symbol`, and `Array`. By updating these definitions, we make it possible for you to change the behavior of these classes (as is required in Exercises 4, 29, 32, and 33).

While the initial basis is being read, calls to `closeLiteralsCycle` and `closeBlocksCycle` fail. We ignore these failures, but once initialization is complete, these cycles are closed by the code in chunk 495c, which catches failures.

494a ⟨*evaluation* 486a⟩+≡ (475a) ◁493c

```
and readEvalPrint (defs : def stream, echo, errmsg) xi =
  let fun processDef (def, xi) =
        let fun continue msg = (errmsg msg; showStackTrace (); resetTrace (); xi)
            val _ = (closeLiteralsCycle xi; closeBlocksCycle xi)
                    handle NotFound _ => ()
            in  evaldef (def, echo, xi)
                handle IO.Io {name, ...} => continue ("I/O error: " ^ name)
                ⟨more read-eval-print handlers 222c⟩
            end
  in  streamFold processDef xi (defs : def stream)
  end
```

The ⟨*more read-eval-print handlers* 494b⟩ include the handlers used in μScheme. We add handlers for the Standard ML exceptions `Subscript` and `Size`, which may be raised by array primitives.

⟨*more read-eval-print handlers* 494b⟩≡

```
| Subscript       => continue ("array subscript out of bounds")
| Size            => continue ("bad array size")
```


10.6.9 Initializing, bootstrapping, and running the interpreter

The first step in creating the initial environment is to add the built-in classes. Each one needs a metaclass to be an instance of. To be faithful to Smalltalk, the subclass relationships of the metaclasses should be isomorphic to the subclass relationships of the classes. This is true for the user-defined classes created with `newClassObject`, but on the built-in classes, we cheat: the metaclasses for `UndefinedObject` and `Class` inherit directly from `Class`, not from `Object`'s metaclass.

⟨*initialization* 494c⟩≡ (475a) 495a▷

```
val initialXi = emptyEnv

fun mkMeta c = mkClass ("class " ^ className c) classClass [] []
fun addClass (c, xi) = bind (className c, ref (mkMeta c, CLASSREP c), xi)
val initialXi = foldl addClass initialXi [ objectClass, nilClass, classClass ]
```

The next step is to read the class definitions in the initial basis. For debugging, it can be helpful to set `echoBasis` to `true`.

495a ⟨*initialization* 494c⟩+≡ (475a) ◁494c 495b▷

```
val echoBasis = false
val initialXi =
  let val defs = reader smalltalkSyntax noPrompts
                 ("initial basis", streamOfList ⟨ML representation of initial basis (automatically generated)⟩)
  in  readEvalPrint
        ( defs : def stream
        , echoBasis
        , fn s => app print ["error in initial basis: ", s, "\n"]
        )
        initialXi
  end
```

Before we can close the cycles, we have to create `VAL` bindings for `true` and `false`. Because the parser prevents user code from binding `true` and `false`, we can't do this in μSmalltalk; the bindings have to be added in ML.

495b ⟨*initialization* 494c⟩+≡ (475a) ◁495a 495c▷

```
local
  fun newInstance classname = SEND (nullsrc, "new", VAR classname, [])
in
  val initialXi = evaldef (VAL ("true",  newInstance "True" ), false, initialXi)
  val initialXi = evaldef (VAL ("false", newInstance "False"), false, initialXi)
end
```

Once we've read the class definitions, we can close the cycles, update the ref cells, and we're almost ready to go. By this time, all the necessary classes must be defined, so if any cycle fails to close, we halt the interpreter with a fatal error.

495c ⟨*initialization* 494c⟩+≡ (475a) ◁495b 496a▷

```
val _ =
  ( closeLiteralsCycle initialXi
  ; closeBooleansCycle initialXi
  ; closeBlocksCycle   initialXi
  ) handle NotFound n =>
      ( app print ["Fatal error: ", n, " is not defined in the initial basis\n"]
      ; raise InternalError "this can't happen"
      )
  | e => ( print "Error binding basis classes into interpreter\n"; raise e)
```

The numeric and collection classes are in the initial basis.

495d ⟨*additions to the μSmalltalk initial basis* 446⟩+≡ ◁482a

```
⟨numeric classes 459a⟩
⟨collection classes 448b⟩
```

The last step initialization is to bind predefined values. The value `nil` is bound here because the parser won't let us create a `val` binding for it. The other values are symbols that are useful for printing, but that can't be expressed using μSmalltalk's literal symbol notation.

496a ⟨*initialization* 494c⟩+≡ (475a) ◁495c 496b▷

```
fun addVal ((name, v), xi) = evaldef (VAL (name, VALUE v), false, xi)
val initialXi = foldl addVal initialXi
  [ ("nil", nilValue)
  , ("lparen",  mkSymbol "(",  ("rparen", mkSymbol ")")
  , ("lbrack",  mkSymbol "[",  ("rbrack", mkSymbol "]")
  , ("newline", mkSymbol "\n"), ("space",  mkSymbol " ")
  ]
```

The function `runInterpreter` takes one argument, which tells it whether to prompt.

496b ⟨*initialization* 494c⟩+≡ (475a) ◁496a

```
runInterpreter : bool -> unit
```

```
fun runInterpreter noisy =
  let val prompts = if noisy then stdPrompts else noPrompts
      val defs =
        reader smalltalkSyntax prompts ("standard input", streamOfLines TextIO.stdIn)
      fun writeln s = app print [s, "\n"]
      fun errorln s = TextIO.output (TextIO.stdErr, s ^ "\n")
  in  ignore (readEvalPrint (defs : def stream, true, errorln) initialXi)
  end
```

10.7 Smalltalk as it really is

Smalltalk carries the principle of "simple but powerful" to a remarkable extreme; the language itself is small and simple, and it is used to define a class hierarchy that is large and powerful. In this section, I show you Smalltalk-80's syntax and some features of Smalltalk-80 that do not appear in μSmalltalk. I also sketch parts of Smalltalk-80's class hierarchy.

type def 467b
evaldef 493a
initialXi 495b
mkSymbol 476c
nilValue 485c
noPrompts 668d
reader 669
readEvalPrint 494a
smalltalkSyntax 705a
stdPrompts 668d
type stream 647a
streamOfLines 648b
VAL 467b
VALUE 467a

10.7.1 The language

Syntax

Smalltalk-80 isn't just a language; it's a complete programming environment. Because this environment provides a graphical user interface for creating classes, Smalltalk-80 has no official ASCII syntax. Smalltalk-80 does have a "publication format," which is used in the Blue Book (Goldberg and Robson 1983), and in this section, that's what I use. As a first example, Figure 10.18 shows the publication format of the FinancialHistory class in this format, as shown in the Blue Book.

Compare this version of FinancialHistory with mine from page 402. The protocol is almost the same as mine, except that I don't redefine the class method `new`.[17] The computations are also the same, but both the abstract syntax and the concrete syntax are different.

- The declarations at the top, giving the class name, superclass, and instance variables, are exactly analogous to the same parts of a class definition in μSmalltalk.

[17] In using `FinancialHistory` as an introductory example, I wanted to avoid `super`. Also, to avoid using the word "receive" in the name of a method, I've changed some of the names used in the Blue Book.

```
class name                          FinancialHistory
superclass                          Object
instance variable names             cashOnHand
                                    incomes
                                    outgoes

class methods
instance creation
    initialBalance: amount
        ↑super new setInitialBalance: amount
    new
        ↑super new setInitialBalance: 0

instance methods
transaction recording
    deposit: amount from: source
        incomes at: source
                put: (self totalDepositedFrom: source) + amount.
        cashOnHand ← cashOnHand + amount
    withdraw: amount for: reason
        outgoes at: reason
                put: (self totalWithdrawnFor: reason) + amount.
        cashOnHand ← cashOnHand − amount
inquiries
    cashOnHand
        ↑cashOnHand
    totalDepositedFrom: source
        (incomes includesKey: source)
            ifTrue: [↑incomes at: source]
            ifFalse: [↑0]
    totalWithdrawnFor: reason
        (outgoes includesKey: reason)
            ifTrue: [↑outgoes at: reason]
            ifFalse: [↑0]
private
    setInitialBalance: amount
        cashOnHand ← amount.
        incomes ← Dictionary new.
        outgoes ← Dictionary new
```

Figure 10.18: Class FinancialHistory, in Smalltalk-80 publication format

- Instead of tagging each individual method as a class method or an instance method, Smalltalk-80 tags groups of methods.

- The Smalltalk-80 definition has information about *subgroups* of methods: "instance creation," "transaction recording," "inquiries," and "private." These subgroups, which have no analog in μSmalltalk, serve as documentation. In Smalltalk-80 the subgroups are called *message categories*.

- Each method definition is introduced by a line set in bold type; this line, which is called the *message pattern*, gives the name of the method and the names of its arguments. A method that expects no arguments has a name composed of letters and numbers. A method that expects one argument has a colon after the name; the name of the method is followed by the name of the argument. When a method expects more than one argument, the name of the method is split into *keywords*: each keyword ends with a colon and is followed by the name of one argument. follows each keyword. This concrete syntax matches the concrete syntax of message sends, as shown below.

  When we talk about methods that have two or more arguments, we name them by concatenating the keywords; thus, the official name of the first message is "deposit:from:."

- The bodies of the methods are indented relative to the message patterns. The body of a method contains a sequence of *statements*; Smalltalk-80 is statement-oriented like C, not expression-oriented like Scheme and μSmalltalk. Statements in a sequence are separated by periods. To return a value from a method, a Smalltalk-80 programmer uses *nonlocal return*, which is written using the up arrow $\uparrow$. A nonlocal return stops the method's evaluation immediately and makes it return a value. A method that doesn't use a nonlocal return is evaluated only for side effects.

- Within a statement, the most obvious departure from μSmalltalk syntax is that in a Smalltalk-80 message send, *the receiver of a message precedes the name of that message*. Where in μSmalltalk we write, in the body of `deposit:from:`,

  ```
  (totalDepositedFrom: self source),
  ```

  in Smalltalk-80 it becomes

  ```
  self totalDepositedFrom: source.
  ```

  When a message is sent with arguments, the receiver also appears first, and each argument follows a keyword that is part of the message name. You can find an example in the definition or deposit:from:, which sends the message at:put:.

Compared with μSmalltalk's syntactic structure of "keywords/receiver/arguments," Smalltalk-80's syntactic structure of "receiver/keyword/argument/..." simplifies the concrete syntax when the argument to a message send is itself another message send. You can decide which form you find more readable,

```
(ifTrue:ifFalse: (includesKey: incomes source)
    [(at: incomes source)]
    [0]))
```

or

```
(incomes includesKey: source)
    ifTrue: [(at: incomes source)]
    ifFalse: [0]
```

Receiver-first concrete syntax is used in most object-oriented languages, including C++, Java, Javascript, Objective C, Python, and Ruby, among others. Interestingly, only Objective C supports Smalltalk's keyword/argument concrete syntax. Several of the other languages do, however, allow you to simulate keyword syntax by passing a dictionary as an argument.

- As in μSmalltalk, even + and − are messages; in Smalltalk-80, messages formed with these and other symbols are *binary messages*: they are sent to a receiver and expect one argument. Binary messages have higher precedence than keyword messages, so fewer parentheses are needed. For example, in the withdraw:for: method, the at:put: message is sent to outgoes. The precedence of + requires the parentheses around self totalWithdrawnFor: reason, and implies that the entire expression (self totalWithdrawnFor: reason) + amount is the second argument to at:put:.
- The "←" symbol is used for assignment.
- The message-passing syntax requires that a Boolean come first, *before* ifTrue:ifFalse:. As in μSmalltalk, the square brackets denote blocks.

Smalltalk-80 uses a lightweight syntax for blocks, even when blocks have arguments. Arguments are written after the opening "[" and before a vertical bar "|"; their names are preceded by colons. Thus, the block we write in μSmalltalk as

```
(block (x) (add: self x))
```

is written in Smalltalk-80 as

```
[:x | self add: x].
```

Smalltalk-80 uses different messages to evaluate blocks, depending on the number of arguments. To evaluate a zero-argument block, send it the value message; to evaluate a one-argument block, send it the value: message; to evaluate a two-argument block, send it the value:value: message; and so on.

Semantics

Smalltalk-80 includes a few features that I have left out of μSmalltalk. For example, Smalltalk-80 has many kinds of built-in literals:

Class	Literal
Integer	230
Character	$a
Float	3.39e-5
Symbol	#sameasours
String	's p a c e s'
Array	#($a $b $c)

Smalltalk-80 also has *class variables*, which are different from instance variables and global variables. A class variable is shared by every instance of its class, and it can be referred to by any method in the class or in a subclass. Class variables can be useful when, for example, it is necessary to keep track of all the objects of a class. Class variables also make global variables unnecessary; because *every* class inherits from Object, the class variables of class Object play the role of global variables. There's nothing magical about class variables; by analogy with class methods, the class variables of class C are simply the instance variables of C's metaclass. If you want to try adding class variables to μSmalltalk, do Exercise 36.

Nonlocal return

Smalltalk-80's return is "nonlocal" because it can stop the evaluation of more than one method. For example, here is an implementation of detect:ifNone: in Smalltalk-80:

```
detect: aBlock ifNone: exnBlock
  self do: [:x | (aBlock value: x) ifTrue: [↑x]].
  exnBlock value
```

When the nonlocal return is evaluated, it terminates not only detect:ifNone:, but also the nested evaluations of do: and ifTrue:.

Primitive methods

In μSmalltalk, a method is either user-defined or primitive. In Smalltalk-80, a method's definition can combine a primitive with user code. In the general case, a Smalltalk-80 method is defined as a primitive method followed by one or more statements:

message-pattern
  <primitive n>
  statements

When the method is dispatched, it executes the primitive numbered n. (In Smalltalk-80, primitives are numbered, not named.) If for any reason the primitive method fails, the *statements* are executed; if the primitive succeeds, its result is returned and the *statements* are ignored.

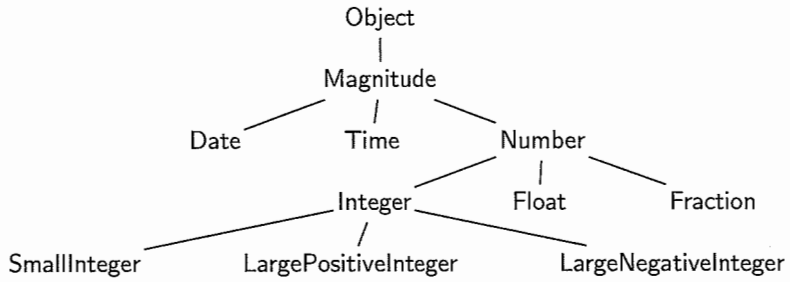
Application programmers never need to write primitive methods; as in μSmalltalk, they are used only to create the predefined classes.

10.7.2 The class hierarchy in Smalltalk-80

Smalltalk-80 comes with many, many class definitions. In this section, we examine the Number and Collection sub-hierarchies of the Smalltalk-80 hierarchy; these general-purpose classes are used by all programmers. There are also many special-purpose classes in the hierarchy, including those related to graphics, compilation, and process scheduling, which are beyond the scope of this book.

Numbers

In Smalltalk-80, the part of the class hierarchy that contains numeric classes looks like this:



I've modeled μSmalltalk's `Magnitude` and `Number` classes on the Smalltalk-80 versions. Smalltalk-80 Numbers respond to more messages than μSmalltalk `Numbers`. Classes `Float` and `Fraction` are likewise similar in both languages. As noted above, Smalltalk-80 provides floating-point notation for literals of class Float.

As in μSmalltalk, Integer is an abstract class, and it has one concrete subclass: `SmallInteger`. Smalltalk-80 provides three concrete subclasses of `Integer`: SmallInteger objects fit into machine words, and LargePositiveInteger and LargeNegativeInteger provide for arbitrary precision. In Smalltalk-80, Integer literals may be arbitrarily long; the class of an integer literal is determined by its magnitude and sign.

As in other dynamically typed languages, the class of the result of an arithmetic operation may be different from the class of its receiver or arguments. What is unusual about Smalltalk is the class of the result may depend upon the *value* of the receiver or argument, not just its class. For example, 10 raisedTo: 2 returns a SmallInteger, but 10 raisedTo: 20 returns a LargePositiveInteger. The Smalltalk-80 rule is to return an object of the *least general* class that can hold the result, where, for example, SmallInteger is less general than LargePositiveInteger, which is less general than Float. That is why 10 raisedTo: 20 does not return a Float. All these coercions are handled within the Numbers hierarchy.

The Number operations make good use of Smalltalk-80's more powerful primitive methods. For example, the definition of + on SmallInteger uses primitive method 1 to do the addition, but if the primitive method fails, e.g., because the addition overflows, the succeeding statements can automatically shift to large-integer arithmetic:

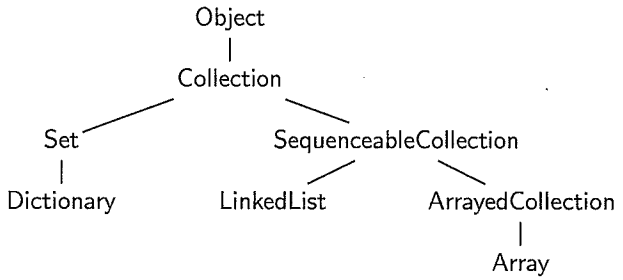
```
+ aSmallInteger
  <primitive 1>
  ↑self asLargeInteger + aSmallInteger

asLargeInteger
  self negative
    ifTrue: [↑self asLargeNegativeInteger]
    ifFalse: [↑self asLargePositiveInteger]
```

The simple coercion asLargeInteger is enough to effect the shift.

Unlike μSmalltalk, Smalltalk-80 supports *mixed* arithmetic; for example, it can add a fraction to an integer. The result of mixed arithmetic uses the "least general" class that is capable of representing the answer.

Collection classes

Smalltalk-80 provides a Collection sub-hierarchy that looks a bit different from the `Collection` hierarchy in μSmalltalk. The μSmalltalk hierarchy is derived from the one in Tim Budd's (1987) Little Smalltalk. Here is a part of the Smalltalk-80 collection hierarchy:



Both the Smalltalk and μSmalltalk collection hierarchies rely on iteration (`do:`) as the fundamental operation.

Reflection

In Smalltalk-80, all aspects of the implementation are themselves exposed as objects. The ability of a language to manipulate its own implementation is sometimes called *reflection*. Reflection it makes it possible to implement class browsers, compilers, debuggers, garbage collectors, and so on in the language itself. Here's a taste of what can be done.

You can get the class of any Smalltalk-80 object by sending it the class message. Once you have a class object, you can add or remove methods, query for particular methods, and call methods by name. You can compile or decompile code associated with methods. You can change the superclass and add or remove subclasses. You can get the names of all the instance variables, which you can also add and remove. You can get information about the representations of instances. The Smalltalk-80 compiler, which is written in Smalltalk, uses these capabilities to read in Smalltalk source code and compile it into new classes and objects.

Even the "active contexts" used to implement message send are represented in the system as objects. This means, for example, that a debugger or trace routine can visit all active contexts and display values of local variables and instance variables, all using the reflection facilities.

μSmalltalk lacks reflective facilities because they introduce more circularities of the kind discussed in Section 10.6.3. For example, if we added a class primitive to μSmalltalk, then not only would every class have to be an object, as we have now, but μSmalltalk *metaclasses* would also have to be objects. These objects could themselves be sent the `class` message, and so on out to infinity. Infinity can be avoided by having metaclasses eventually loop back to itself, so that sending class messages eventually starts producing the same objects again. Every time I try to understand exactly how Smalltalk-80 accomplishes this feat, my head explodes.

10.8 Summary

Smalltalk helped object-oriented languages take over the world. Object-oriented seems very effective for certain types of problems, including simulation, graphics, and user interfaces. Although classes and objects were first introduced in Simula 67, the compelling case for these ideas was made by Smalltalk. Apple uses many of the programming-environment ideas developed for Smalltalk, and Apple's devices are programmed using the object-oriented language Objective C, a close relative of Smalltalk. The wild ideas that Alan Kay thought might "amplify human reach" are now central to mainstream programming.

10.8.1 Glossary

Abstract class A class that has no instances. Abstract classes are defined in order to supply method definitions that can be inherited by subclasses.

Block A Smalltalk object that represents a function. These values are "first-class" in the sense that they can be passed as arguments to methods and returned as results, and can be assigned to variables. Blocks can have any number of arguments. Blocks are used to implement control structures.

Class The basic programming unit in Smalltalk. A class encapsulates the representation of a certain kind of object and defines the operations on objects of that kind.

Class variable A variable that has one occurrence in a class (regardless of the number of instances of that class). Subclasses can refer to a class's class variables.

Inheritance The feature of Smalltalk whereby a class, being declared as a subclass of another, defines objects that have all the instance variables, and respond to all the messages, of that other class. New components can be added to the representation by the subclass, and new methods can be defined and inherited ones redefined. This is also known as **single inheritance**.

In **multiple inheritance**, which is not supported by Smalltalk, a class can be declared as a subclass of *several* classes, inheriting representation and methods from all of them.

Instance variable A variable of which each instance of a class has its own version. The instance variables of a class define the representation of objects of that class.

Message The name of a method. Since Smalltalk has **overloading**—the ability to define the same message in several places—it becomes important to distinguish between *message* and *method*.

Method The Smalltalk term for a function defined within a class (and all functions in Smalltalk are defined in classes). Methods always have at least one argument, called the **receiver** of the message. In Smalltalk message-sending syntax, the message (i.e. method name) follows the expression defining the receiver.

Method search Inheritance is implemented by having the interpreter search for the method corresponding to a given message. This search begins in the class of the receiver of the message (with the exception of messages sent to `super`), and continues up the hierarchy until a definition for the message is found.

Object A value of a class; also called an **instance**. In Smalltalk, everything is an object.

Overloading The defining of a single message name in more than one class, or, more generally, the use of a name in different ways. In this more general sense, many languages allow some form of overloading; for example, Pascal overloads the name "+", using it for both integer and floating-point addition. In Smalltalk, overloading is particularly important, as it permits the definition of polymorphic functions. C++ has both Smalltalk-style overloading resolved by method-search, and "static" overloading resolved by the compiler based on the types of the function's arguments.

Polymorphism The use of a single function for different types of data. Since in Smalltalk the only notion of "type" is "protocol to which an object responds," any piece of code can be applied to any object that responds to the set of messages sent to it in that code; and since a message can be defined in more than one class, a piece of code can be applied to objects from different classes.

Protocol The set of messages to which objects of a given class can respond.

self A special variable that can be used in method definitions to refer to the receiver of the message.

Subclass A class which is placed underneath another class in the inheritance hierarchy.

super A special variable that can be used as the receiver of a message in an expression, with the following meaning: the object that receives the message is actually `self`, but the method search begins in the superclass of the class in which the expression occurs.

Superclass A class which is placed above another class in the inheritance hierarchy.

10.8.2 Further Reading

The the basic reference on Smalltalk-80 is "blue book," by Goldberg and Robson (1983), which is so called because of its predominantly blue cover. The blue book introduces the language and class hierarchy, and it also explains a lot about the implementation and the virtual machine on which it is based. In designing μSmalltalk, I drew heavily on Goldberg and Robson, especially for the treatment of blocks. The "red book" (Goldberg 1983) describes the programming environment and its implementation. There is also a "green book," which describes some of the history of Smalltalk (Krasner 1983). More recently, Ingalls et al. (1997) describe Squeak, a free, portable implementation of Smalltalk that is written in Smalltalk itself; more information is available at `www.squeak.org`.

The August 1981 issue of Byte Magazine (Byte 1981) contains several articles by members of the Smalltalk group, presenting a fairly thorough introduction to the language and the programming environment. The paperback by Kaehler and Patterson (1986) is a simplified, concrete introduction, taking the user through a series of sessions with the Smalltalk system, developing different solutions to the Towers of Hanoi problem. If you want to tackle something simpler than Smalltalk-80 but more ambitious than μSmalltalk, Budd (1987) describes "Little Smalltalk," which is nearly identical to Smalltalk-80, but which lacks some extras, especially around graphics. Budd's book is easier reading than the blue book, and you can get an interpreter, written in C, from its author. The excellent book by Beck (1997) discusses proper coding style for Smalltalk; it will help you make your Smalltalk code idiomatic.

In the end, no one book tells you everything you need to know to become an effective Smalltalk programmer. The truly effective Smalltalk programmer has mastered the class hierarchy, and to gain such mastery, you need to read and write code.

Object-oriented programming has become wildly popular, and many, many programming languages either identify themselves as object-oriented or include object-oriented features. Among these languages, Simula 67 was the very first to provide classes and inheritance (Dahl and Hoare 1972; Birtwistle et al. 1973). Several languages add object-oriented features to an imperative foundation. C++ (Stroustrup 1997) adds object-oriented features—and many others besides—to C. Objective C is another object-oriented extension of C; it is described by Cox (1986). Schmucker (1986) describes an object-oriented extension of Pascal that runs on the Macintosh. Modula-3 (Nelson 1991) is an object-oriented extension of Modula-2. It in turn inspired the development of the scripting language Python (van Rossum 1998) and the general-purpose application language Java (Gosling, Joy, and Steele 1997). Another scripting language, Ruby (Flanagan and Matsumoto 2008), is more directly inspired by Smalltalk.

Not all object-oriented languages are extensions of earlier languages; some are, like Smalltalk, designed from scratch. Examples are Owl (Schaffert et al. 1986) and Eiffel, described in the excellent book by Meyer (1992), which also gives an interesting defense of object-oriented programming in general; see also Meyer 1997. Of the languages above, only Objective C, Python, and Ruby share with Smalltalk the property of being dynamically typed; the others are statically typed.

The Lisp community got on the object-oriented bandwagon early. The Flavors system (Moon 1986) was first developed by MIT's Lisp-Machine group in 1979; it includes multiple inheritance and takes a rather different view of the object-oriented paradigm. CommonLoops and the Common Lisp Object System (Bobrow et al. 1986, 1988) are object-oriented extensions to Common Lisp.

There are many books on computer simulation, running the gamut from highly theoretical to quite pragmatic. The pragmatic books by Payne (1982) and by Schriber (1974) also include introductions to the best-known special-purpose simulation languages. Goldberg and Robson (1983) and Budd (1987) include extended examples of simulations in Smalltalk; Birtwistle et al. (1973) do the same for Simula 67.

10.9 Exercises

Highlights

This chapter has three sets of exercises that I especially like:

- Implementing an efficient priority queue (Exercises 16 and 17) is relatively easy, and it shows an object-oriented approach to a classic problem in data structures.
- Smalltalk's method dispatch might seem inefficient. But many call sites dispatch to the same method over and over—a property that clever implementations exploit. Exercise 39 asks you to implement a simple *method cache* and measure its effectiveness. The results will surprise you.
- Exercises 31 to 33 are the most ambitious exercises in this chapter. You implement a classic abstraction: full integers of unbounded magnitude. I've split the project into three stages: natural numbers, signed large integers with arithmetic, and finally "mixed arithmetic," which uses small machine integers when possible and transparently switches over to large integers when necessary. You will boost your object-oriented programming skills, and you will understand, from the ground up, an implementation of arithmetic that should be available in every civilized programming language.

Guide to all the exercises

Exercises 1 to 4 give you practice defining and using new classes and methods. Exercise 1 asks for a standalone random-number class; do it if you're not entirely confident about defining new classes. Exercises 2 to 4 ask for various improvements in sequenceable collections, especially arrays. Do some of these if you want to get a feel for collections.

Exercise 5 asks you to build a new kind of collection: bags, which are also called multisets. The exercise will familiarize you with collections, but the real challenge is to build the new collection using as little new code as possible. Reusing old code is what inheritance is for.

Exercise 6 asks you to implement a more powerful array abstraction that not only provides constant-time access to elements but also can grow and shrink at need. This is a classic data structure, and it is built into several programming languages, including Clu, Icon, and Lua. Exercise 7 follows up by asking you to use this data structure to implement `List`. These exercises are good if you want to see how data-structure design plays out in an object-oriented setting.

Exercise 9 calls your attention to an inefficiency in the `FinancialHistory` class and asks you to fix it. Do this exercise to start to familiarize yourself with some of the more unusual methods defined on collections. Exercise 10 asks you to fix a small inefficiency in class `Collection`.

Exercises 11 to 13 ask you to improve the `List` class defined in Section 10.5.3. Like the array exercises, these exercises are good for getting familiar with collections. And Exercise 13 will push you to start thinking like an object-oriented programmer: you need to consider not only what code to write, but also with what classes that code should be associated.

Exercises 14 and 15 ask you to improve dictionaries in particular and keyed collections more generally.

Exercises 16 and 17 ask for two different implementations of the class `PriorityQueue` used in discrete-event simulations. Exercise 16 is the simpler implementation; Exercise 17 asks for an implementation that is slightly more complex, but much faster.

Exercises 18 to 22 invite you to explore discrete-event simulation in more depth. Exercise 18 suggests a number of ways to make the robot-lab simulation (Section 10.4) more realistic. Exercise 19 asks you to improve the resource-handling code so that it can be written once and used for many simulations. Exercise 20 asks you to develop better ways of generating streams of events. Exercise 21 asks you to create new `Student` objects using a *factory object* rather than a class. Finally, Exercise 22 asks you to repair a defect in the design of the `Simulation` class.

Exercises 23 to 25 ask for new primitives on arrays, Booleans, and symbols.

Exercise 26 asks you to investigate the possibility of compile-time type checking for Smalltalk. You will have to connect the ideas in this chapter with the ideas in Chapter 6.

Exercises 27 to 29 explore the numeric classes. Exercises 27 and 28 ask you to reason about overflow. Exercise 29 is a bigger challenge: it asks you to arrange for seamless mixed arithmetic combining integers and fractions. Tackling Exercise 29 will help you appreciate just how nice a dynamic language like Smalltalk can be, and it will also give you some idea of how much programming effort it takes to provide nice facilities.

Exercise 30 is a brief diversion into the `Magnitude` classes: you implement dates and times, which are magnitudes but not numbers.

In Exercises 31 to 33, you implement natural numbers (Exercise 31) and arbitrary-precision integers (Exercise 32), then finish by arranging that if a computation on small integers overflows, μSmalltalk transparently switches over to large integers. In these three exercises, you get to build true integer arithmetic—a feature that is central to any civilized programming language. You also get to see both the challenges and the benefits of object-oriented programming: while your simple algorithms may be "smeared out" over the methods of several different classes, you will find satisfying opportunities for inheriting and reusing methods.

In most of this book, we look at inference rules and code, or we ask you to translate inference rules into code. Exercise 34 asks you translate code into inference rules: look at the code for `findMethod` and produce inference rules for the judgment form $m \triangleright c \mathbin{@} impl$.

Exercise 35 shifts your attention to the μSmalltalk interpreter, asking you to add a primitive method to class `Class` that provides access to the class's name.

Exercise 36 asks you to add class variables to μSmalltalk.

Exercise 37 asks you to add primitive Booleans and `if` to the interpreter, and to compare costs with our implementation, which implements Booleans and conditionals without any primitives.

Exercise 38 asks to to explore an alternative way to implement `super`.

Exercise 39 asks you to extend the μSmalltalk interpreter with a *method cache*, which is a simple way to speed up method dispatch.

Exercise 40 asks you to implement a message-send profiler and to use it to identify hot spots in code.

Exercise 41 is a coda to Exercises 31 to 33. It asks you to change the μSmalltalk interpreter so that the transparent switchover from small integers to large integers can be implemented more realistically and efficiently.

Learning about the language

1. Define a class `Rand` having the following protocol:

   - Class method `new:` creates a new random-number generator in which the seed is the argument, which should be an integer.
   - Instance method `next` returns the next random number and updates the seed.

   Generate the random numbers as on page 92.

2. Define class `Interval` as a subclass of `SequenceableCollection`. This class represents finite subsets of the integers defined by arithmetic progressions. In other words, its objects are sets of the form $\{n, n+k, n+2k, \ldots, n+mk\}$ for some $n, k > 0$ and $m \geq 0$. An interval is initialized by giving n, m, and k, and is immutable. Its instance protocol is the same as other `SequenceableCollections`, except that it lacks `add:`. The class protocol should provide two initializing methods: `from:to:` and `from:to:by:`. (`from:to:by: Interval` n $n+mk$ k) answers an interval containing the progression named above; (`from:to: Interval` n $n+m$) uses $k = 1$.

3. Use `Interval` to implement `associationsDo:`, `collect:`, and `select:` on arrays, and to re-implement `do:`.

4. Add to class `Array` a class method `from: aCollection` that makes an array out of the elements of another collection.

5. Define the class `Bag`, whose objects are sets with multiple occurrences of elements (also called *multisets*). The protocol is the same as `Set`, but the generators should return as many occurrences of an element as have been added, and the method `count:` should give the number of occurrences of `item`. For example:

```
-> (val B (mk Bag))
-> (add: B 1)
-> (add: B 1)
-> (first B)
1
-> (next B)
1
-> (next B)
nil
-> (count: B 1)
2
```

   You should try to find a place in the `Collection` hierarchy to insert `Bag` so as to write as little new code as possible; do not change the existing hierarchy.

6. Fixed-size arrays are a nuisance. Create a `VariableArray` class that inherits from `SequenceableCollection` and that represents arrays with indices from n to m, inclusive. It should also support the following new messages.

   - The class method `new` should create a new, empty array with $n = 1$ and $m = 0$.
   - The instance method `addlo:` should decrease n by 1 and add its argument in the new slot. The instance method `addhi:` should increase m by 1 and add its argument in the new slot. The method `add:` should be a synonym for `addlo:`.
   - The instance method `remlo` should remove the first element, increase n by 1, and answer the removed element. The instance method `remhi` should remove the last element, decrease m by 1, and answer the removed element.

   All these operations should take constant time in the normal case, as should `at:`.

   Here are some suggestions about representation:

   - Use a fixed-size array `a` to hold the elements.
   - Keep n and m in instance variables, and use another instance variable `first` to keep track of the position within `a` of element n.
   - Allow the elements to "wrap around" `a`. That is, be prepared for elements $n, \ldots n+k$ to be in elements `first`, $\ldots$ `first`$+k$ of `a`, and the remaining elements $n+k+1, \ldots, m$ to be in elements $1, \ldots m-(n+k)$ of `a`, where `first`$+k =$ (`size a`).
   - If you have to add an element when the array is full, i.e., when $m - n + 1 =$ (`size a`), allocate a new `a` that is twice the size of the old one, and copy the elements of the old one into the new one.

7. The `VariableArray` class from Exercise 6 could be used to implement `List`.

   (a) Estimate the space savings for large lists, in both the average case and the worst case.

   (b) Write an implementation.

8. Section 10.2.4 says that sending `new` to a class should answer a newly allocated object whose state respects the private invariants of the class. For example, a private invariant of the class `FinancialHistory` is that `incomes` refers to a dictionary. But the definition of `FinancialHistory` in Figure 10.2 simply inherits `new` from `Object`, and `new` returns an *uninitialized* `FinancialHistory` object. The class definition does not include `new` because in such an early example, I did not want to introduce the technique of sending messages to `super`. Fix the problem such that `new` creates a history with a zero balances, by sending `initialBalance:` to `self`. You will also need to change the `initialBalance:` class method to use `super`.

9. In the implementation of `FinancialHistory` in Figure 10.2, the `totalDepositedFrom:` and `totalWithdrawnFor:` methods first test for the presence of a key, and only then send `at:`. It is inefficient to make two probes into the same data structure; study the `KeyedCollection` protocol in Figure 10.8, and find a way to rewrite the `totalDepositedFrom:` and `totalWithdrawnFor:` methods so they make only a single probe.

10. Redefine the `detect:ifNone:` method on `Collection` so as to avoid the final `if`.

11. Implement `removeLast` for class `List`.

12. Implement `associationsDo:` for class `List`. Arrange to share code with the `do:` method; you may find a private method useful.

13. If the index is out of range, the `at:put:` method on class `List` silently produces a wrong answer. What an embarrassment! Try the following:

    (a) Supplement the existing implementation with an explicit range check.

    (b) Throw away the existing code, and in its place devise a solution that delegates the work to classes `Cons` and `ListSentinel`.

    (c) Compare the two solutions above, say which you like better, and why.

14. Implement `remove:ifAbsent:` for dictionaries. Is this method useful? If not, what other method would be more useful?

15. Increase code reuse in the methods of `KeyedCollection` by introducing a private `detectAssociation:ifAbsent:` method that takes two blocks as arguments. Explain why this might be a good thing even if it makes the code harder to understand. *Hint: what about implementations that use hash tables?*

    Aggressive reuse often makes the code harder to follow, because the implementation is divided into more small pieces, which are probably distributed over the definitions of several classes. If every bit of duplicated code is abstracted out into its own method, every nontrivial computation appears to be composed of many methods, each of which does almost nothing. The learning curve for such a system can we quite high. This problem is characteristic of aggressively object-oriented systems.

    Argue whether your change to `KeyedCollection` is an improvement.

16. The discrete-event simulation described in Section 10.4 requires a priority queue, whose protocol is given in Figure 10.15 on page 431. Use the variable-size arrays from Exercise 6 on page 509 to implement class `PriorityQueue`:

    (a) As your representation, use a variable-size array that holds a sequence of `Associations`. In each `Association`, the value represents an event, and the key represents the time at which the event is scheduled to occur.

    (b) Maintain the invariant that the array is sorted by event time. You can then implement `deleteMin` using `remlo`, and you can implement `at:put:` by using `addhi:` and then sifting down the new element into its new position in the array.

    (c) Prove that this implementation takes constant time for `deleteMin` and $O(n)$ time for `at:put:`, where n is the number of elements in the queue.

17. If we're implementing a priority queue, we can do better than $O(n)$ time for insertion. You can implement a faster algorithm if you store the queue's elements in an array which is indexed from 1 to n and which satisfies the following invariant:

$$\forall k. a[k] \leq a[2k] \wedge a[k] \leq a[2k+1],$$

    whenever $2k \leq n$ and $2k+1 \leq n$.

    (a) Prove that the invariant implies that $a[1]$ is the smallest element of the array.

(b) Prove that removing the last element maintains the invariant.

(c) If the first element is replaced by an arbitrary element, the invariant can be re-established by the following procedure:

let $k = 1$
while ($2k \leq n$ and $a[k] > a[2k]$) or ($2k + 1 \leq n$ and $a[k] > a[2k + 1]$) do
  swap $a[k]$ with the smaller of $a[2k]$ and $a[2k + 1]$
  replace k with $2k$ or $2k + 1$, whichever was used to swap

If an arbitrary element is added at the end, the invariant can be established by similar procedure involving repeated swapping with $a[\lfloor \frac{k}{2} \rfloor]$.

(d) Use these facts to implement a priority queue. You can use the extensible arrays from Exercise 6, or you can implement a simpler extensible array that grows and shrinks only at the right-hand side.

(e) Measure the effect on simulation time.

18. There are a number of ways we could improve the robot-lab simulation described in Section 10.4.

(a) Professor S gets a big grant and buys three new robots, increasing the number in the lab to 5. Reimplement the `Lab` class so it can easily represent a lab containing 5 robots. Make sure that when robots wear out or future robots are acquired, the code will be easy to update. (Hint: the initial basis includes class `Set`.)

(b) Define a new simulation `VerboseRobotLabSimulation`, which prints a message each time a student leaves the lab. The message should identify the student, the time of arrival, and the time of departure. Don't touch any existing code. Remember `super`.

(c) Modify the model to allow for balking (when a student arrives and finds the waiting line longer than five students, the student leaves immediately) and to account for context-switching time (if a student has to relinquish a robot before having finished, that student now needs five more minutes).

(d) For each student finishing, compute his or her *time-waiting ratio*: time spent in the lab divided by time spent using robots. (To represent the ratio, use `Fraction` or `Float`.) At the end of a simulation, report on the largest time-waiting ratio suffered during that simulation. Does this result contradict the average waiting time measure?

Try to solve this problem without modifying any existing code but only by defining subclasses.

(e) Student arrivals should be random. A process of random arrivals occurring at a fixed rate is called a *Poisson process.* In a Poisson process, the probability density function for interarrival times Δt is an exponential $e^{-\lambda \Delta t}$, where λ is the arrival rate measured in students per minute. If you have a way of generating random floating-point numbers U over the unit interval $[0.0, 1.0]$, you can compute a suitably distributed Δt by using the equation

$$\Delta t = \frac{-\ln U}{\lambda}.$$

Implement a `PoissonEveryNMinutes` class which uses random numbers to deliver *random* interarrival times with an expected rate of $\frac{1}{N}$ students per minute. To compute the natural logarithm in μSmalltalk you can either use an approximation method suited to computing the log of a number between 0 and 1, or you can modify the interpreter to add a primitive logarithm based on the Standard ML function `Math.ln`, which operates on floating-point numbers.

19. In the discrete-event simulation described in Section 10.4, robots are *fungible.* That is, one robot is as good as any other robot, so as long as a `Student` object gets a robot, it doesn't matter which one. Simulations turn out to be full of fungible resources: examples include luggage carts, Boeing 747s, gallons of gasoline, and twenty-dollar bills. There is no reason that every new simulation class should have to implement code to manage fungible resources—it should be done once in the superclass.

    Design and implement methods on class `Simulation` that allow simulation objects to manage arbitrary collections of named, fungible resources. You might consider some of the following methods:

    - A method that requests a single resource (or N units of resource) by name.
    - A method that returns resources.
    - A method that makes a resource name *known* to the simulation. Attempts to request or return resources with unknown names should cause run-time errors.
    - Methods that tell the simulation to create or destroy resources.

    In addition, you will have to expand the protocol for simulation objects so that any simulation object can be granted resources by name.

    Your implementation should generalize the code in the robot-lab simulation: if a simulation object requests an available resource, the request should be granted right away; if a simulation object requests an unavailable resource, the object should be put onto a queue associated with the resource.

    To check your work, you can reimplement the robot-lab simulation using your new methods.

20. In the discrete-event simulation described in Section 10.4, the implementation of streams should offend you: there is no composition and no reuse. Design and implement a library of stream classes that offer the following functionality:

    (a) Implement a superclass `Stream` that includes the collection methods `select:`, `reject:`, and `collect:`. Method `next` should be a subclass responsibility.

(b) Implement a subclass stream s in which something occurs every n minutes. That is, sending `next` always answers n.

(c) Given a stream s and a limit N, produce a new stream s' that such that repeatedly sending `next` produces the first N elements of s and afterward answers only `nil`.

(d) Given two streams s_1 and s_2, produce a new stream s such that repeatedly sending `next` to s produces first all the elements of s_1, followed by all the elements of s_2.

(e) Given two streams s_1 and s_2, produce a new stream s such that repeatedly sending `next` to s produces alternating elements of s_1 and s_2 (that is, s_1 and s_2 "take turns").

(f) Use your library to reimplement the streams used in the discrete-event simulation.

21. In the discrete-event simulation described in Section 10.4, when we have a new model of students' needs, we have to create a new subclass of class `Student`. Creating these classes is tedious, and this coding style makes it unnecessarily hard to, for example, read needs from a file. Address these problems by creating a single class `StudentFactory`, such that

    - To create `StudentFactory`, you supply a stream of needs to a class method `new:`.
    - An *instance* of class `StudentFactor` can respond to a `new` message, which it does by pulling the "time needed" from its stream, then creating and answering a new instance of `Student` with that need.

    You will need to create a subclass of `Student` that works with the `StudentFactory`.

    The idea of using an object to create other objects is so popular that "Factory" is used as the name of a *design pattern.*

22. The `Simulation` class in Section 10.4.2 is badly designed: although the `startUp`, `proceed`, and `finishUp` methods provide a handy way to organize initialization and finalization, they can't actually be used by clients, because if the event queue happens to be empty, it's not safe to call `proceed`. Repair this defect by changing class `Simulation`. Change the implementation, and if necessary, change the protocol as well.

23. Class `Array` inherits `at:ifAbsent:` from `SequenceableCollection`. This implementation costs time linear in the size of the array. Without defining any new primitives, build a new implementation that takes constant time:

    (a) Define a new, private method on class `Array`, which should provide access to the to the primitive `arrayAt:`. You might as well call this private method `arrayAt:`.

    (b) Remove the definition of `at:` from class `Array`, so it inherits `at:` from `SequenceableCollection`.

    (c) On class `Array`, `at:ifAbsent:` using `arrayAt:` and the other methods of class `Array`.

24. Change the predefined Boolean classes as described in ⟨*potential additions to the μSmalltalk initial basis* 447b⟩. Can you find an application for which there is there a measurable speedup?

25. Define a class `SymbolString` which represents a string of symbols.

    (a) Define a `^` method that concatenates a string or a symbol to a string of symbols. Should the class be mutable or immutable?

    (b) Now extend `Symbol` so that `^` is defined on symbols as well. Should the class be mutable or immutable?

26. When a message is sent to a Smalltalk object, we can't tell at compile time what method will be used to answer. The dynamic nature of method dispatch makes compile-time type checking difficult. Two enlightening discussions of this problem are by Johnson, Graver, and Zurawski (1988), who describe a type-checking extension to Smalltalk, and Meyer (1988, sections 11.3 and 11.4), who describes the type-checking rules of Eiffel. Read these papers and report on the type-checking problem in object-oriented languages. Discuss both efficiency and correctness.

27. In class `Fraction`, could redefining `<` in terms of `-` reduce the possibility of overflow? Prove your answer.

28. As defined in Section 10.5.4, floating-point addition might overflow. Re-implement floating-point addition so that it neither overflows nor loses precision unnecessarily. For example, you shouldn't lose precision adding 0.1 to 1. Here are some suggestions:

    - Make each exponent as small as it can be without overflowing.
    - Then use the larger of the two exponents.

29. This problem explores improvements in the built-in numeric classes, to try to support mixed arithmetic.

    (a) Arrange the `Fraction` and `Integer` classes so you can add integers to fractions, subtract integers from fractions, multiply fractions by integers, etc. You should be able to achieve all this with just two lines of code.

    (b) Change the `Integer` class so you can add fractions to integers. This requires much more work than part 29a. You might be tempted to change the `+` method to test to see if its argument is an integer or a fraction, then proceed. A better technique is to use *double dispatch*: the `+` method does nothing but send a message to its argument, asking the argument to add `self`. The key is that the message encodes the type of `self`. For example, using double dispatch, the `+` method on `Integer` might be defined this way:

514 ⟨*double dispatch* 514⟩≡

```
(method + (aNumber)
    (addIntegerTo: aNumber self))
```

The classes `Integer`, `Float`, and `Fraction` would all then define methods for `addIntegerTo:`.

(c) Complete your work on the `Integer` class so that all the arithmetic operations, including = and <, work on mixed integers and fractions. Minimize the number of new messages you have to introduce. *Hint: you may find it useful to change or remove some existing definitions on* `Integer` *and* `SmallInteger`.

(d) Finish the job by making `Fraction`'s methods answer an integer if the denominator of the fraction is 1.

30. In μSmalltalk, the only interesting `Numbers` are `Magnitudes`. Define more `Magnitudes`!

(a) Define `Date` as a subclass of `Magnitude`. A `Date` is given by a month, day, and year. Define a `+` method on `Date` that adds a number of days to the date. This method should know how many days are in each month, and it should also know the rules for leap years.

(b) Define `Time` as a subclass of `Magnitude`. `Time` objects represent time on a 24–hour clock. Define a `+` method that adds minutes.

31. Create class `Natural`, which should be a subclass of `Magnitude`. A value of class `Natural` should represent a natural number *of any size*. We suggest a representation using two instance variables: `degree` and `digits`. The representation invariant should be as follows: `digits` should be an array containing at least `degree` + 1 integers, each of which lies in the range $0 \leq x_i < b$, where b is a constant that you choose.[18] If we imagine that the array `digits` contains an array of coefficients x_i, where $0 \leq i \leq$ `degree`, then the abstraction function says that the array represents natural number X, where

$$X = \sum_{i=0}^{\text{degree}} x_i b^i.$$

In addition to the protocol for `Magnitude`, instances of class `Natural` should respond to the following protocol:

`+ aNatural`	Answer the sum of the receiver and the argument.
`- aNatural`	Answer the difference of the receiver and the argument, or fail with a run-time error if this difference is not a natural number.
`* aNatural`	Answer the product of the receiver and the argument.
`div: aNatural`	Answer the largest natural number whose value is at most the quotient of the receiver and the argument.
`isZero`	Answer `true` if the receiver is zero, and `false` otherwise.
`subtract:withDifference:ifNegative: aNatural diffBlock negativeBlock`	Subtract `aNatural` from the receiver to obtain difference d. If the difference is a natural number, answer `(value diffBlock` d`)`. If the difference is negative, answer `(value negativeBlock)`.

The class `Natural` itself should respond to the message `new: anInteger` by creating a new instance whose value is equal to the argument. It may be easiest to create a list of digits, then build `digits` by sending `from:` to `Array` (Exercise 4).

Private class method for class `Natural`:

`base`	Answers b.

Private instance methods for class `Natural`:

`digit: anIndex`	Upon receiving `digit:` i, answer x_i. Should work for any nonnegative i, no matter how large.
`digit:put: anIndex aDigit`	On receiving `digit:put:` i y, mutate the receiver, making $x_i = y$. Although `Natural` is not a mutable type (and therefore this method should never be called by clients), it can be quite useful to mutate individual digits while you are constructing a new instance.
`digits: aSequence`	Take a sequence of x_i and initialize `digits` and `degree`.
`doDigits: aBlock`	For i from zero to `degree`, send `value` i to `aBlock`.
`trim`	Set `degree` on the receiver as small as possible, and answer the receiver.
`degree`	Answer the `degree` of the receiver.
`decimal`	Answer a `List` containing the *decimal* digits of the receiver, most significant digit first. N.B. unless $b = 10$, this method cannot simply return the elements of `digits`. The `add:` method on lists can help with this method, and this method can help with `print`.
`makeEmpty: aDegree`	Set `digits` to an array suitable for representing natural numbers of the specified degree. (Also change the `degree` of the receiver to `aDegree`.)
`set:plus: x y`	Overwrite the digits of the receiver to hold `x` + `y`, where `x` and `y` are natural numbers. If the receiver does not have enough digits to hold the sum, cause a checked run-time error.
`set:minus: x y`	Overwrite the digits of the receiver to hold `x` − `y`, where `x` and `y` are natural numbers. Answer the borrow out, which will be 0 if `x` $\geq$ `y` and 1 otherwise. If the receiver does not have enough digits to hold the sum, cause a checked run-time error.

Figure 10.19: Suggested private methods for class `Natural`

You may also find it useful to implement the private methods in Figure 10.19. Some algebraic identities can help with the arithmetic. The identity for addition is:

$$\begin{aligned} X+Y &= (\sum_i x_i b^i) + (\sum_j y_j b^j) \\ &= \sum_{\substack{i=0 \\ \text{max degree of } x \text{ and } y}} x_i b^i + y_i b^i \\ &= \sum_{\substack{i=0 \\ \text{max degree of } x \text{ and } y}} (x_i + y_i) b^i \end{aligned}$$

The identity above almost gives you the digits of the sum, except that you might violate the representation invariant; it is possible that $x_i + y_i \geq b$. You can deal with this violation by using a carry bit. Subtraction is similar, except the possible violation is $x_i - y_i < 0$, and you can use a borrow bit.

The identity for multiplication is:

$$\begin{aligned} X \times Y &= (\sum_i x_i b^i) \times (\sum_j y_j b^j) \\ &= \sum_i \sum_j x_i y_j b^{i+j} \end{aligned}$$

Here, the possibility that $x_i y_i \geq b$ risks violating the representation invariant, so you have to split this value over two digits. Use the law $zb^i = (z \bmod b)b^i + \lfloor z/b \rfloor b^{i+1}$.

Long division can't be implemented with a simple identity; Brinch Hansen (1994) can help.

32. Using sign-magnitude representation, implement large signed integers. You will find it helpful to represent the magnitude using an instance variable of class `Natural`, while you encode the sign in the class itself. You can achieve this encoding by creating classes `LargePositiveInteger` and `LargeNegativeInteger`, both of which inherit from the *abstract* class `LargeInteger`.

517 ⟨*large integers* 517⟩≡

```
(class LargeInteger Integer
  (magnitude)
  (class-method withMagnitude: (aNatural)
      (magnitude: (new self) aNatural))
  (method magnitude () magnitude)
  (method magnitude: (aNatural)
      (set magnitude aNatural)
      self)
  (class-method new: (anInteger)
     (if (negative anInteger)
        [(magnitude: (new LargeNegativeInteger) (new: Natural (negated anInteger)))]
        [(magnitude: (new LargePositiveInteger) (new: Natural anInteger))]))
  (method asLargeInteger () self)
  (method isZero () (isZero magnitude))
  (method = (anInteger) (isZero   (- self anInteger)))
  (method < (anInteger) (negative (- self anInteger)))
)
```

[18] Letting $b = 10$ will be inefficient, but it will help you debug.

Do *not* create instances of class `LargeInteger`; instead, create instances of classes `LargePositiveInteger` and `LargeNegativeInteger`, which you should define as subclasses of `LargeInteger`. Define whatever remaining methods are needed to answer everything from the `Integer` protocol. You will probably find it most useful to focus on these methods: `print`, `negative`, `positive`, `strictlyPositive`, `negated`, `+`, `*`, and `div:`.

- You can implement `negated` fairly easily by creating an instance of the opposite sign using the same magnitude.
- For addition, use *double dispatch*. That is, extend every integer class by adding the three methods `addSmallIntegerTo:`, `addLargeNegativeIntegerTo:`, and `addLargePositiveIntegerTo:`. Then use these methods to redefine the `+` method. For example, the definition of class `LargePositiveInteger` should include the following method definition:

```
(method + (anInteger) (addLargePositiveIntegerTo: anInteger self))
```

- Add an `asLargeInteger` method to class `SmallInteger`.
- Your implementations of `*` and `div:` should also use double dispatch.

33. Change the definition of class `SmallInteger` such that if arithmetic overflows, the system automatically moves to large integers. Furthermore, make sure that arithmetic between small and large integers works transparently as needed.

    You will need the following new primitives, which are already implemented as shown below:

> `add:withOverflow: aSmallInteger ovBlock`
> Compute the sum of the receiver and the argument `aSmallInteger`. If this computation overflows, answer `ovBlock`; otherwise answer a block that will answer the sum.
>
> `sub:withOverflow: aSmallInteger ovBlock`
> Compute the difference of the receiver and the argument `aSmallInteger`. If this computation overflows, answer `ovBlock`; otherwise answer a block that will answer the difference.
>
> `mul:withOverflow: aSmallInteger ovBlock`
> Compute the product of the receiver and the argument `aSmallInteger`. If this computation overflows, answer `ovBlock`; otherwise answer a block that will answer the product.

    These primitives require a level of indirection; instead of answering directly with the result, they answer a block capable of returning the result. This property is needed to arrange for the proper execution of the recovery code, but it makes the use of these primitives a bit strange. Here's an example, which is the analog of the Smalltalk-80 code on page 502:

518 ⟨*example use of primitives with overflow recovery* 518⟩≡

```
(class SmallInteger SmallInteger ; redefine SmallInteger
   ()
   (method + (aNumber) (addSmallIntegerTo: aNumber self))
   (method addSmallIntegerTo: (anInteger)
      (value (add:withOverflow: self anInteger
                                [(+ (asLargeInteger self) anInteger)])))
   (method add:withOverflow: primitive add:withOverflow:)
)
```

The implementations of the primitives are easy; we try to build a block containing the result, but if the computation overflows, we answer the overflow block instead.

519a ⟨*primitive methods for remaining classes* 480f⟩+≡ (475a) ◁484

```
fun withOverflow binop ((_, NUM n), [(_, NUM m), ovflw]) =
      (mkBlock ([], [VALUE (mkInteger (binop (n, m)))], emptyEnv, objectClass)
       handle Overflow => ovflw)
  | withOverflow _ (_, [_, _]) =
      raise RuntimeError "numeric primitive with overflow expects numbers"
  | withOverflow _ _ =
      raise RuntimeError "numeric primitive with overflow expects receiver + 2 args"
```

519b ⟨*primitive methods* :: 480a⟩+≡ (492c) ◁482f

```
primMethod "add:withOverflow:" (withOverflow op + ) ::
primMethod "sub:withOverflow:" (withOverflow op - ) ::
primMethod "mul:withOverflow:" (withOverflow op * ) ::
```

A familiar way to get large integers is to compute factorials. Factorials don't make a great test, because they only test multiplication, and they never multiply two large integers together. But they still make a good start, so here is a `Factorial` class you can use:

520 ⟨*large-arithmetic transcript* 520⟩≡

```
-> (class Factorial Object
     ()
     (class-method printUpto: (limit) (locals n nfac)
         (set n 1)
         (set nfac 1)
         (while [(<= n limit)]
            [(print n) (print #!) (print space) (print #=)
                                  (print space) (println nfac)
             (set n (+ n 1))
             (set nfac (* n nfac))])))
-> (use mixed.solution)
-> (use bignum.solution)
-> (printUpto: Factorial 20)
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
13! = 6227020800
14! = 87178291200
15! = 1307674368000
16! = 20922789888000
17! = 355687428096000
18! = 6402373705728000
19! = 121645100408832000
20! = 2432902008176640000
nil
```


Learning about the interpreter

34. In Section 10.6.2, we present the judgment form $m \triangleright c @ impl$, which describes method dispatch. We didn't give any proof rules for this form, but there is a function `findMethod` in chunk 486b which is supposed to implement it. Using the implementation as a guide, write inference rules for the judgment.

35. Our design of the collection hierarchy requires every concrete collection class to implement a `printName` method. But the μSmalltalk interpreter already stores the class name in every ML value of type `class`. Provide direct access to this knowledge by adding a predefined `name` method to μSmalltalk class `Class`. Use this predefined method to eliminate as many definitions of `printName` as possible, without changing the observable behavior of the interpreter. (It is fair game to change existing definitions of `printName` as needed.) How many definitions of `printName` must you keep, and why?

36. The robot-lab simulation in Section 10.4 puts the currently active simulation in a global variable. In Smalltalk-80, this information would be in a *class variable* available only to simulation objects. Add class variables to μSmalltalk. The syntax for class definitions becomes:

    def ::= `(class` *subclass-name superclass-name*
    `[(class-variables` {*class-variable-name*}`)]`
    `(`{*instance-variable-name*}`)`
    {*method-definition*}`)`

    Add a new field of type `string list` to `CLASS` and `CLASSD`. The initial `rho` in `evalMethod` should be built from the *class* variables, and instance variables should be added afterwards, to ensure that class variables are visible after instance variables but before global variables.

    Test your implementation of class variables by rewriting the robot-lab simulation: add a new class `SimulationObject` which holds the currently active simulation as a class variable.

37. This question explores the consequences of making Booleans non-primitive objects.

    (a) Would a primitive Boolean representation `BOOL`, analogous to `NUM`, make things any faster?

    (b) Suppose `eval` directly implements sending `if` to a Boolean, just as it directly implements sending `value` to a block. Would that make things any faster? What other Boolean methods should be implemented directly in the interpreter?

    (c) Implement these two optimizations and measure the results. Compare also against the improved basis of Exercise 24 and the method cache of Exercise 39.

38. An alternative version of `super` is this: when a message `m` is sent to `super`, send it to `self`, but start the method search in the superclass of `self`'s class. Compare this with the correct version of `super`. Find an example from this chapter in which the incorrect version does the wrong thing.

    You may find it expedient to implement the incorrect version, the better to understand the difference.

39. Because Smalltalk does everything with method dispatch, method dispatch has to be fast. Since all computation, even integer arithmetic, is done by message-passing, unnecessary overhead will kill performance. An effective way to reduce this overhead is *inline caching of method addresses*, as described by Deutsch and Schiffman (1984). The idea works like this: with each `SEND` node in an abstract-syntax tree, associate two words of memory, one giving the class of the last object to receive that message, the other the address of the method that was executed the last time the message was sent. Whenever the message is sent, this one-element cache is consulted; if the class of the current receiver is the same as the class of the last receiver (as stored in the cache), then the method stored in the cache is the one to be executed, and the method search can be side-stepped. If the class of the current receiver is different, then undertake the usual method search, and update the cache. This technique saves time if there is a high probability that the receiver of a message will be of the same class as the most recent receiver of that message; research has suggested that this probability is about 95%. Method lookup can be a serious bottleneck; Conroy and Pelegri-Llopart (1982) report that adding a method cache made the early Berkeley Smalltalk system run 37% faster.

    (a) Extend the μSmalltalk interpreter with method caches.

    - Add a field of type `(int * method) ref` to the `SEND` node in the abstract syntax. This reference cell should hold a pair representing the unique identifier of the last class to receive the message at this call site, plus the method found on that class.
    - Change the parser to initialize each cache. You can use -1 as the initial identifier, since -1 is guaranteed not to be the identifier of any class.
    - Change function `ev` in chunk 490 to use the cache before calling `findMethod`, and to update the cache afterward (provided a method is found). You will probably find it useful to write a new function `findCachedMethod` of type `(int * method) ref * name * class -> method`.

    (b) Add code to gather statistics about hits and misses, and measure the cache-hit ratio for a variety of programs.

    (c) Find or create some long-running μSmalltalk programs, and measure the total speedup obtained by caching.

40. Implement a message-send profiler that counts the number of times each message is sent to each class. Don't forget the `value` message. Use this profiler to identify hot spots in the simulation code.

41. Change the way μSmalltalk handles primitive methods so that the kinds of indirection used in Exercise 33 are not necessary. That is, make it possible to add two small integers without creating any blocks, but still recovering from overflow. You will have to extend the definition of primitive methods to include optional "recovery code," and you will have to arrange for the evaluator to execute this recovery code if the primitive raises an ML exception. This change will make μSmalltalk more consistent with the way Smalltalk-80 implements primitive methods, as well as making it more efficient. For ideas, consult the Smalltalk blue book (Goldberg and Robson 1983).