Chapter 2
A Tour of Squeak

1 Basic Rules of Smalltalk

The basic rules of Smalltalk can be stated pretty simply.

- Everything is an object. This is by far the most important rule in Smalltalk.
- All computation is triggered through message sends. You send a message to an object, and something happens.
- Almost all executable Smalltalk expressions are of the form **<receiverObject> <message>**.
- Messages trigger *methods* where the mapping of message-to-methods is determined by the receiving object. Methods are the units of Smalltalk code. You can think of a method as being like a function or procedure in your favorite programming language. It's the place where the computation occurs.
- Every object is an *instance* of some *class*. 12 is an instance of the class SmallInteger. ‘abc’ is an instance of the class String. The class determines the data and behavior of its instances.
- All classes have a parent class, except for the class Object. The parent class defines data and behavior that is *inherited* by all of its children classes. The parent class is called a *superclass* and the children classes are called *subclasses*.

Let's look at an example piece of Smalltalk code:

```
| anArray anIndex aValue |
aValue := 2.
anArray := Array new: 10.
1 to: 10 do:
  [:index |
  anArray at: index
              put: (aValue * index)].
anIndex := 1.
[anIndex < anArray size]
  whileTrue:
  [Transcript show:
        'Value at: ',(anIndex printString),
        ' is ',
```

```
        (anArray at: anIndex) printString ; cr.
    anIndex := anIndex + 1.]
```

This looks pretty similar to code that you might see in any programming language. You see assignment statements, expressions, creation of an array object, a structures that looks like a **for** loop, and a structure that looks like a **while** loop. You may notice the lack of type declarations and some seemingly odd syntax. For the most part, your intuition about the meaning of these pieces will be correct. But the real semantics are different than what you may expect (and in many ways, simpler and more consistent) because of the basic rules of Smalltalk.

- *Everything is an object.* This rule means that **aValue := 2** does not actually mean "Set the value of 'aValue' to integer 2" but instead means "Set the variable aValue to point to an **SmallInteger** object whose value is 2." (Be careful of the case of things here -- Smalltalk **is** case sensitive, and array is not the same as **Array**.) There is no type associated with a variable. Variables just point to objects, and everything is an object. If the next line had aValue being assigned to a string (e.g., **aValue := 'fred the string')** or even a window, it's all the same to Smalltalk. The variable aValue would still point to an object, and everything is an object.
- *All computation is triggered through message sends.* This rule means that even the pieces above that look like special constructs, like **1 to: 10 do:** and **[anIndex < anArray size]** are just message sends.
- *Almost all executable Smalltalk expressions are of the form* <***receiverObject***> <***message***>. This one can lead to some surprises when coming to Smalltalk from more traditional programming languages. **1 to: 10 do: []** is a message send to the object **1**! The message **to: do:** is a message understood by **Integers**! 10 and the block of code (statements contained in square brackets) following **do:** are actually arguments in the message. Consider expressions like **2 + 3**. In the semantics of Smalltalk, this is a message send of **+** with the argument of **3** to the object **2**. While it may seem unusual, such adherence to a single standard mechanism has proven to be amazingly powerful!
- *Messages trigger methods*. Each of the messages mentioned above (**to: do:, whileTrue:, +**) trigger *methods* which are Smalltalk code units. You can view the implementation of control structures and operators—and even change them!

It's important to note the difference between messages and methods. In many languages (e.g., C, Pascal, Fortran, Cobol), the function name defines the code to be executed. If you execute **foo(12)** in any of these

languages, you know exactly what is going to be executed. Smalltalk is a kind of language that uses *late-binding*. When you see the message **printString**, you actually do not know what is going to be executed until you know the object that is being sent the message. **printString** always returns a string representation of the object receiving the message. **20 printString**, **32.456 printString**, and **FileDirectory default printString** actually have very different implementations, even though they are all responding to the same message. They also provide the same functionality—they return a printable, string representation of the receiving object. If the receiver object is a variable, then it's not possible at compile-time to figure out which method to invoke for the given message. The decision of which method will execute for the given message is made at runtime (hence, *late-binding*), when the receiver object is known.

Having the same message perform approximately the same functions on different data is called *polymorphism*. It's an important feature of object-oriented languages, as well as other languages. Addition is polymorphic in most languages. Addition in the form **3 + 5** is actually a very different operation at the machine's level than **3.1 + 5.2**, but it's the same message or operation at the human's level. What's nice about most object-oriented languages is that you the programmer can define your own polymorphic messages.

The programmer is not specifying a piece of code when she sends the message **printString** to some object. Rather, she is specifying a *goal*: To get a printable, string representation of the object. Since this goal may be implemented differently depending on the kind of object, there will be multiple *methods* implementing the same *message*. Programming in terms of goal shifts the focus of programming to a higher level, out of the bits and into the objects.

- *Every object is an instance of some class.* Since the class is where the definition of the instance's behavior resides, it's very important to find the class of the receiver object to figure out how a message will be interpreted.
- *All classes have a parent object.* Consider the code above **(aValue * index)**. **aValue** in this example is bound to a **SmallInteger**, and **SmallIntegers** know how to multiply (*****). But we might also ask **aValue** if it's positive (**aValue positive**), a test which returns true or false. **SmallInteger's** do not know how to tell if they're positive, but **Numbers** do, and **SmallIntegers** are a kind of **Number**. (Strictly, **SmallInteger** is a subclass of **Integer**, which is a subclass of **Number**.) We can also ask **aValue** what the maximum is of itself or another number (e.g., **aValue max: 12**). **max:** is a message understood by **Magnitude**, not by **SmallInteger** or **Number**—and

Number is a subclass of **Magnitude**, and thus **aValue** inherits what **Magnitude's** know. **Date** and **Time** are two other subclasses of **Magnitude**, so it's possible to get the maximum between any two dates or any two time instances, but it may not be possible to do arithmetic on them.

```
Magnitude
        Character
        Date
        Number
                Float
                Fraction
                Integer
                        SmallInteger
        Time
```

Figure 1: The Hierarchy of Classes Below Magnitude

2 Doing "Normal" Things in Squeak

Much of your programming in Squeak involves the same kind of programming that you've done in any other language: Variables, control structures, and manipulating numbers, strings, and files. A good way of starting with Squeak is to explain how you do these "normal" operations.

If you want to try out some pieces of this code (which you're encouraged to do!), you can try these expressions and code fragments in a workspace. Start up Squeak by opening the image with the executable. (In UNIX, you can type the name of the executable then the name of the image file; on Windows or on a Mac, just drag the image file onto the executable.) Click the mouse button down anywhere on the desktop and hold to bring up the Desktop or World Menu (Figure 2). Release the mouse with **Open…** highlighted. Choose **Workspace** (Figure 3).

Figure 2: The Desktop (or World) Menu

In a workspace, you can type code, select it, then execute it and print the result. In Figure 3, the workspace contains **3 + 4**. On UNIX, you do Control-P; on Windows, Alt-P; and on Macs, Apple/Command-P. This is referred to by Smalltalkers as a **PrintIt** operation. The result is that the expression is evaluated and the resultant object is printed.

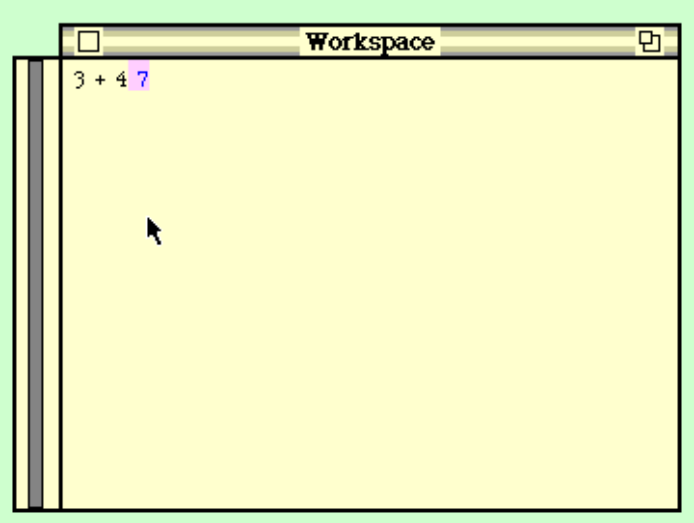

Figure 3: An example workspace

2.1 Variables and Statements

Variables can be essentially any single word that starts with a letter. The assignment operator is := or can also be ← (which is typed as an underscore character in the basic Squeak font). The value of an assignment (what PrintIt displays) is the right hand side of the assignment.

```
aVariable := 12.
aVariable ← 'Here is a String'.
```

Any Smalltalk expression can work as a statement. Statements are separated with periods. A perfectly valid sequence of Smalltalk statements is:

```
1 < 2.
12 positive.
3 + 4.
```

There is a "scope" associated with variables in Smalltalk. If you created **aVariable** in your workspace with the above example, your variable's scope would be that workspace. The variable would be accessible within the workspace, as long as the workspace is open, but nowhere else in the system. If you did a PrintIt on

```
myVariable := 34.5.
```

you would get 34.5. If you then did a PrintIt on:

```
myVariable + 1
```

you would get 35.5. The variable exists within the workspace.

You can also create variables that are local only to the execution of a group of Smalltalk statements, as in this line from the example at the beginning of the chapter.

```
| anArray anIndex aValue |
```

The beginning of code segments can hold local variable declarations. These variables will only exist for the duration of the code being executed.

As a matter of Smalltalk style, variables, method names, and other local names all begin with lowercase letters. Globals, which includes all class names, begin with an uppercase letter. This style rule is enforced at various places in the system. For example, if you PrintIt

```
MyVariable := 29.
```

you'll get a dialog box asking you if you really did want to declare a global variable (Figure 4). You can declare the variable global, but Smalltalk assumes that you were actually trying to reference an existing global variable, so it offers a selection of potential alternatives based on what you typed.

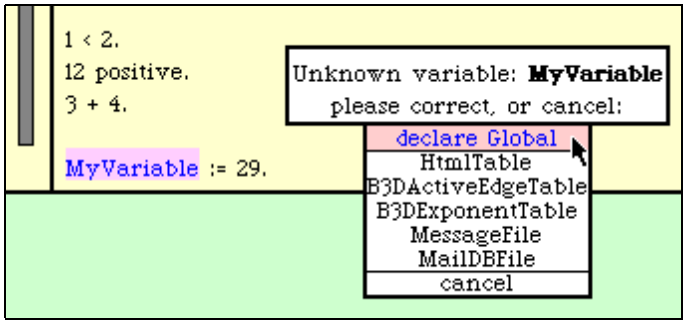

Figure 4: Dialog on Declaring a Global Variable

You've probably noticed that the variable declarations don't say anything about the *type* of the variables: integer, float, arrays, public or private, static or dynamic, etc. Smalltalk is essentially a *type-less* programming language. Everything in Smalltalk is simply an object. Any variable can reference any object. All *collection* objects (e.g., arrays, sets, bags, ordered collections, etc.) can contain any kind of object. There is no distinction between public and private, as there is in C++ or Java, for data or methods.

You may be wondering what happens in Smalltalk when you try to evaluate something that depends on type, such as **3 + 'fred'** (adding the string 'fred' to the integer 3). The error that you get in this particular instance is "**At least one digit expected here**". If you track down the debugging stack (which is explained later in this chapter), you find that what failed is that the string 'fred' did not understand a message which numbers understand. The way that types fail in Smalltalk is that an object does not understand a message. But on the other hand, making **3 + 'fred'**

actually work is to simply teach Strings to respond to the appropriate messages. The flexibility of the system is enormous.

2.2 Control Structures

We've already seen basic Smalltalk expressions, which is obviously the simplest form of control: Just list one expression after another.

One of the "normal" things that programmers often want to do is to print out results somewhere. Workspace code can't normally print back out to the Workspace, but there is a window accessible via the global variable **Transcript** that can be easily printed to. To open a Transcript, choose **Open…** again from the Desktop Menu, and then select **Transcript**. You can display things to the Transcript by sending it the message **show:** with some string.

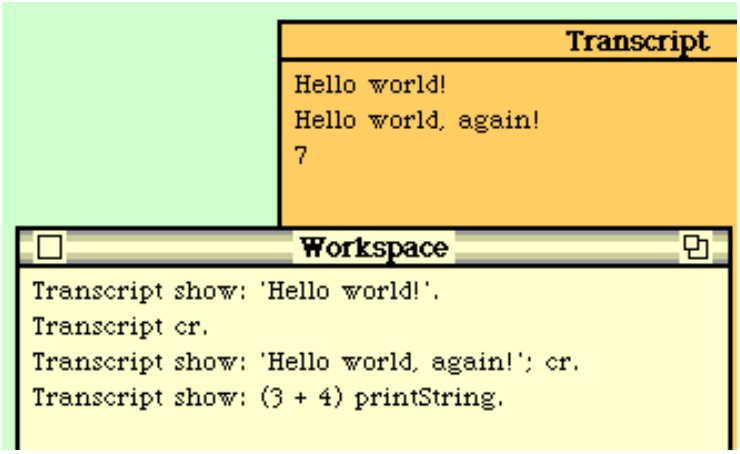

Figure 5: Transcript example

In the example in Figure 5, you see a string being printed to the Transcript. The **cr** message generates a carriage return on the Transcript. The next **show**: will print on the line below. We also see a *message cascade*. A semi-colon can separate a series of messages to the same receiver (**Transcript** in this case). We also see an integer being converted to a printable string, then printed to the Transcript.

All the control structures that you might expect to be in a "normal" language are present in Smalltalk.

```
"if...then"
a < 12 ifTrue: [Transcript show: 'True!'].
```

(Go ahead and PrintIt on the above example.) The first thing to notice is the comment in double quotes at the top of the example. Double quotes delimit comments in Smalltalk.

ifTrue: is a message sent to boolean values. **a < 12** will return either **true** or **false**. That object will then receive the message **ifTrue**: and a *block* of statements in square brackets.

> ***Cautionary Note***: There are objects defined in Smalltalk **true** and **false**. There are also objects **True** and **False**. True and False are the classes, and true and false are the instances of those classes (respectively). **True** and **False** are still *objects*—you can send messages to them. But they understand different messages than the instances **true** and **false**. **True ifTrue: [Smalltalk beep]** will only generate an error. **true ifTrue: [Smalltalk beep]** will beep.

The square brackets define a kind of object called a *block*. A block can be sent messages, or can even be assigned to variables. It's a first class object, like any string or number.

```
"if...then...else"
((a < 12) and: [b > 13])
ifTrue: [Transcript show: 'True!']
ifFalse: [Transcript show: 'False!'].
```

The above example demonstrates an **ifTrue:ifFalse:** which would be an *if-then-else* in a more traditional programming language. The order doesn't matter: There is an **ifFalse:ifTrue:** message for boolean objects, too. You also see a logical *and* in this example. **and:** is a message understood by booleans. It takes a block that will be evaluated if the receiver object is true, that is, it does *short-circuit*. There is also an **or:** message defined for booleans.

The outer set of parentheses is necessary in this example. Without them, Smalltalk would interpret the message very differently. **(a < 12)** would be sent the message **and:ifTrue:ifFalse:**, which of course, is not defined.

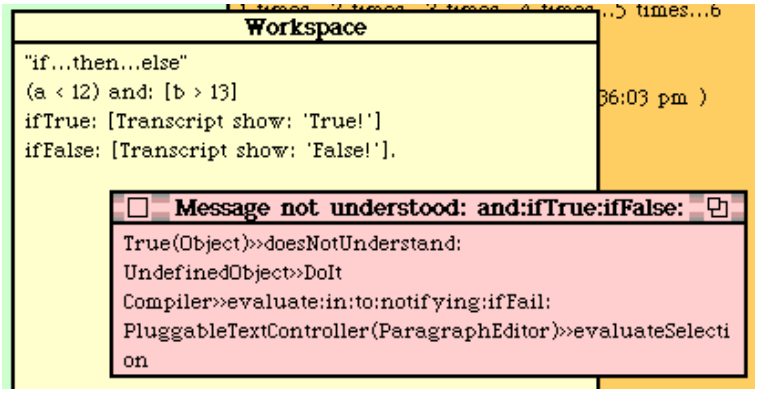

Figure 6: Error notifier resulting from removing the outer parentheses

```
"A while loop"
a ← 1.
```

```
[a < 10] whileTrue:
        [a := a + 1. Transcript show: '9 times...'].
```

This example shows a traditional *while* loop. Both **whileTrue**: and **whileFalse**: are defined in Squeak. Note that the test is a block (enclosed in square brackets), and the body of the *while* loop is also a block. The multiple statements inside the body block are separated by periods.

```
"timesRepeat"
9 timesRepeat: [Transcript show: '9 times...'].
```

A **timesRepeat**: isn't in most programming languages, but is pretty useful. Sometimes, you want something to happen a certain number of times, but you don't need the index variable of a *for* loop.

```
"for loop -- variable could be anything"
1 to: 9 do: [:index | Transcript show:
                (index printString),' times...'].
1 to: 9 do: [:i | Transcript show:
                (i printString),' times...'].
```

We refer to these two messages as **to:do:**. The arguments (the number and the block) are just interspersed amongst the pieces of the message (called the *selector*). Here we see two different **to:do**: loops (a *for* loop in other languages). The only difference between them is a change in the index variable name. A vertical bar separates the definition of the index variable from the rest of the statements in the body of the loop.

2.3 Literals, Numbers, and Operations

What goes on the right side of an assignment is a very rich set of possibilities. Basically, any expression which returns a value (which is always an object) is valid on the right side of an assignment. Literals are certainly valid expressions.

Example	Meaning
12	An integer (in this example, because it's less than 32K, a **SmallInteger**).
34.56	A floating point number (instance of **Float**).
$a	The **Character**, lowercase A.
'a'	The string with the single character lowercase A in it.
#(12 'a' $b)	A literal array with three elements in it: The integer 12, the string 'a', and the character lowercase B.

"a"	This actually means absolutely nothing to Smalltalk—anything inside of double quotes is considered a comment. You can intersperse comments anywhere in your code to help explain it to others or to yourself when you forget what your code means.

SideNote: As in any other programming language, Smalltalk arrays only hold collections of the same kind of element. They are *homogeneous* collections. Smalltalk arrays only hold objects.

A whole set of infix numeric operations (called *binary messages* because they involve two objects) are also available in creating expressions.

Operation	Meaning
4 + 3	Addition
32.3 – 5	Subtraction
65 * 32	Multiplication
67 / 42	Division. The result here is the **Fraction** object **67/42**. Send the fraction the message **asFloat** to get a decimal value.
10 // 3	Quotient, truncating toward negative infinity. Result here is 3.
10 \\ 3	Remainder, truncating toward negative infinity. Result here is 1.

Beyond literals and infix operations lay a vast collection of textual messages. Some of these are *unary*, which means that they take no arguments. Other messages are *keyword messages* where each *selector* ends with a colon ($:) which means that they take arguments. Here are a few examples:

Example	Meaning
(-4) abs	Absolute value. Returns integer 4.
90 sin	Sine of 90 radians. Returns 0.893996663600558
anArray at: 5	Returns whatever object is at position 5 in **anArray**.

$a asUppercase	Returns the character uppercase A
10 // 3	Quotient, truncating toward negative infinity. Result here is 3.
10 \\ 3	Remainder, truncating toward negative infinity. Result here is 1.

The order of precedence is:

- Things in parentheses are evaluated first.
- Unary messages are next.
- Binary messages (infix operators) are next.
- Keyword messages are last.

2.4 Strings and Arrays

Strings and arrays, as in many languages, are similar to one another in Squeak. Strings and arrays respond to some similar messages, because they have a common ancestry in terms of the hierarchy of classes. They both inherit from the class **SequenceableCollection**.

```
Collection
      SequenceableCollection
            ArrayedCollection
                  Array
                        WeakArray
                  Array2D
                  ByteArray
                  FloatArray
                  IntegerArray
                        PointArray
                  SoundBuffer
                  String
                        Symbol
                  Text
            Interval
            LinkedList
                  Semaphore
            MappedCollection
            OrderedCollection
                  SortedCollection
      Set
            Dictionary
                  SystemDictionary
```

Figure 7: A Portion of the Collection class hierarchy

Strings can be created literally with single quotes, but you can also create them with a variety of commands. Here are three ways to create the exact same three character string. We can create it literally. We can create it using the message **with:with:with:** (up to six **with**:'s are understood). We can create a three character **String** and then fill it with the appropriate characters, position by position.

The last statement in the below example is unusual. It's a return. Up-arrow says to return this value. If you select all of those lines, beginning with the **String new:** line, the return will make sure that the value of the whole collection is **aString** when you PrintIt. Without that last line, the value of the whole collection of lines is the last **at:put:**, and the value of an **at:put:** is the value being put, in this case, **$c**.

```
"A literal string"
'abc'
```

```
"Using with:with:with:"
String with: $a with: $b with: $c

"Creating a blank string then filling it."
aString := String new: 3.
aString at: 1 put: $a.
aString at: 2 put: $b.
aString at: 3 put: $c.
^aString
```

That latter example is not quite in traditional Smalltalk style. Typically, Smalltalkers don't create explicit sizes too often, unless one is very, *very* sure of the size. Since most strings have a tendency to grow, strings are generally created without a specific length. Here's an alternative way do to the same thing. In the below example, we add characters in two different ways. In the first, we use the concatenation operator, a comma (**$,**). The concatenation character takes an argument of a string, so the character must be converted to a string with **asString**. In the latter two, we put the new characters at the end of the string with **copyWith**:. We must reassign **aString** each time because these operators create a new string.They don't modify the existing string.

```
"Creating a blank string then filling it."
aString := String new.
aString := aString , $a asString.
aString := aString copyWith: $b.
aString := aString copyWith: $c.
^aString
```

Strings do not expand their length in Squeak. If you want to replace a sequence in a string with a longer or shorter sequence, you need to make a copy of it as you do the replacement.

```
'squeak' copyReplaceAll: 'ea' with: 'awwww'
        "Returns: 'squawwwwk'"
```

Most of the above messages are not specific to Strings. Rather, they're defined higher in the Collections class hierarchy, so they're available to arrays as well. Here are the same four methods for creating an identical array.

```
"A literal array"
#(12 'b' $c)

"Using with:with:with:"
Array with: 12 with: 'b' with: $c

"Creating a blank array then start filling it."
anArray := Array new: 3.
anArray at: 1 put: 12.
anArray at: 2 put: 'b'.
anArray at: 3 put: $c.
^anArray

"Creating a blank array then start filling it."
anArray := Array new.
```

```
anArray := anArray , #(12).
anArray := anArray copyWith: 'b'.
anArray := anArray copyWith: $c.
^anArray
```

There are many operations in common with both arrays and strings. We can access components of each with **at**:. We can execute a block over each element of the array or string with **do**:. We can create a new string or array from evaluating a block to each element with **select**: They share these operations in common with all **Collection** subclasses. They also share operations from their superclasses **SequenceableCollection** and **ArrayedCollection**.

Example	Value
#(12 43 'abc' $g) at: 2 **'squeak' at: 2**	**at:** provides access to elements. Returns **43** and **$q** respectively.
#(12 43 'abc' $g) do: [:element | Transcript show: element printString]. **'squeak' do: [:character | Transcript show: character printString].**	**do:** evaluates the block for each element of the array or string.
#(12 43 55 60) select: [:number | number even] **'squeak' select: [:letter | letter isVowel]**	**select:** evaluates the block for each element, and if the block returns true, will include the element in a new, returned string or array. Returns **(12 60)** and **'uea'**, respectively.

There are many operations that **Collections** such as arrays and strings share, besides the few examples above. You should look through the **Collections** class (and its subclasses) to find useful messages, using the tools described in Section 3. There are four general categories of messages that Collections understand.

- Messages for adding elements, such as **add**: (to add an element) and **addAll**: (to add a whole Collection instance into another).
- Messages for removing elements, such as **remove**: and **removeAll**:
- Messages for testing elements, such as **isEmpty** (to test if a Collection instance is empty), **includes**: (to test for the existence of a given element), and **occurencesOf**: (to count the number of a given element in a collection.)

- Messages for enumerating elements, such as **do**: and **select**: above, but also **reject**: (to collect only the elements that do *not* match a given block), **detect**: (to find the first element that matches a block), and **collect**: (to apply a block to each element of an array and return a collection of the *values* from applying the block).

2.5 Files

Files are manipulated in Squeak via the **FileStream** class. A instance of **FileStream** is opened on a given file, and then access to that file is permitted as a **Stream**.

A **Stream** is a powerful kind of object. It allows access or creation of a large data structure one element at a time. It reduces memory demands by not requiring the large data structure to be resident in memory all at one time.

Create a **FileStream** by opening it on a file with **fileNamed**:. The default, if you don't specify a complete path, is to create a file in the same directory as the current image.

```
aFile ← FileStream fileNamed: 'fred'.
aFile nextPutAll: 'This is a test.'.
aFile close.
```

You can read the file by, again, opening a FileStream on it. There are a couple of ways of manipulating files. The first is just to read the whole thing in as a String, which can be useful for novices who know strings but not streams. **contentsOfEntireFile** will return a string with the file's contents, and then will close the file.

```
aFile ← FileStream fileNamed: 'fred'.
^aFile contentsOfEntireFile
```

Finally, you can also read a file element by element, by sending **next**: to the stream. For a text file, each element is a character.

```
aFile ← FileStream fileNamed: 'fred'.
[aFile atEnd] whileFalse:
        [Transcript show: aFile next printString].
```

Which prints: **This$ is$ a test$**

3 Doing "Object" Things in Squeak

But if Squeak were yet another C or Pascal with an unusually consistent syntax, it would hardly be interesting. Squeak is much more than that, in several different ways. Some of the ways in which Squeak is different are simply due to Squeak being interpretive in nature. The compiler is always available to you, e.g., **Compiler evaluate: '3 + 4'** returns 7 from a PrintIt.

Squeak's strength lies deeper than just its interpretive nature.This section introduces some of the powerful *language* features that were only briefly touched upon in the previous sections. In the sections to come, the *environment* of Squeak is introduced, and how you use that environment to learn Squeak.

3.1 Blocks

Unlike many other programming languages, blocks in Squeak are not just syntactic sugar that are gobbled up by the compiler. Blocks are really objects. (Again, *everything* is an object in Smalltalk.) They can be held in variables, and they can be passed as arguments. You can write code that will create and return blocks.

You can assign a block to a variable just as you would assign any other object to a variable. If you PrintIt on this statement, you will assign a block to the variable aBlock, but what will print won't look like much that makes sense to you. (The printout will look pretty strange—you can just ignore it for now.)

```
aBlock ← [Smalltalk beep].
```

Now, if you ask this block for its **value**, you will hear the beep. Do a PrintIt on this statement.

```
aBlock value.
```

We have also seen blocks that take an argument. Remember the blocks in the **to:do:** and **select:** messages? Those messages don't require a special syntax—they use ordinary blocks that accept arguments. We can create blocks-taking-arguments and store them in variables, too.

```
anArgumentBlock ← [:x | x + 1].
anArgumentBlock value: 5.
```

If you PrintIt on the above, you'll get **6** printed. We can create blocks that take many arguments. Besides **value** and **value**:, blocks also understand messages **value:value:** and **value:value:value:**

Let's consider an example statement from the beginning of the chapter.

```
1 to: 10 do:
   [:index |
   anArray at: index
               put: (aValue * index)].
```

This statement is primarily a keyword message **to:do:** to the receiver object, integer 1. The message takes two arguments, the number 10 and the block of code, delimited by square brackets. The block of code is evaluated within the method **to:do:**, with an argument passed in. The input argument is bound to the *local variable* **index** (it could be named

any valid variable name) in this block. The rest of the block is then executed. In this case, there is only a single statement, which fills each element of **anArray** with twice the value of its index (since **aValue** is set to 2 at the beginning of the example).

We can actually look at the implementation of **to:do:**. It's defined in the class **Number**, which is a *superclass* of **Integer**. The below is called a *method*. It's the actual implementation of the control structure **to:do:**. **stop** and **aBlock** below are the arguments to the method. You see that the method creates a local variable, **nextValue**. **nextValue** is originally set to **self**, which is a special variable that is bound to the receiver object. In the above example, **self** is integer 1. Then there is a **whileTrue**: loop that says while **nextValue** isn't at the stop value, the block takes its value with the **nextValue**. **nextValue** then increments.

```
to: stop do: aBlock
        "Evaluate aBlock for each element of the interval (self to:
stop by: 1)."
        | nextValue |
        nextValue := self.
        [nextValue <= stop]
                whileTrue:
                        [aBlock value: nextValue.
                        nextValue := nextValue + 1]
```

3.2 Variables and Memory

Variables in Smalltalk are different than in many other languages. Variables are not objects *per se*. They are also not just memory locations. Variables *always* point to objects. An uninitialized variable is said to point to **nil**. Any reference to a variable is always a reference to the underlying object. Unlike C or other languages where pointers can be manipulated, the variable itself can never be manipulated in Smalltalk.

The pointer-to-objects nature of Smalltalk variables also means that you can easily, even accidentally, have more than one variable point to the same object. PrintIt on the following:

```
a ← #(1 2 3).
b ← a.
a at: 2 put: 75.
^b
```

The result is **#(1 75 3)**. (Actually, the PrintIt just shows **(1 75 3)**, but it's actually an array.) In this example, **a** points to a literal array, **#(1 2 3)**. **b** is then set to **a**, which means, it points to the same object. When **a**'s second element is changed, **b**'s second element is changed. If we wanted **b** to have a duplicate of **a**'s array, we could say **b ← a copy**. (If **a** was a complex object with internal instance variables that you *also* wanted to copy, you would use **deepCopy**.)

This raises the question of how one would find out if two variables point to the same object, or just have the same values. If **a** has the same value as **b**, **a=b** will return true. But only if **a** and **b** are *actually* the same object will **a==b** return true. One **=** tests for equality, but **==** tests for *equivalence*.

For the most part, all memory management is automatic in Smalltalk. You cannot explicitly release memory. Instead, memory is allocated as needed and released when there are no further references to the memory. The process of reclaiming unused memory is called *garbage collection*, and it occurs in the *background* while other processing is going on. The programmer doesn't see memory allocation nor reclamation, nor does even the user see a pause for garbage collection when moving the mouse or clicking on buttons. The programmer just creates objects as needed. The programmer never sees an empty pointer reference nor a memory fault, which is the real benefit of Smalltalk garbage collection.

Garbage collection occurs when an object has nothing else pointing at it. If you have a workspace in which you have created several variables, all those variables point to objects which cannot be reclaimed by garbage collection. When you close the workspace, the workspace will be reclaimed, as will all the objects that those variables pointed at. Garbage collection doesn't happen immediately, though. Rather, it happens when an object is being allocated and not enough memory

3.3 Creating Classes, Instances, and Methods

By the way, Squeak is an object-oriented programming language, as is Smalltalk that Squeak is based on. You can create classes in it, and instances of those classes. You can define data that all instances of the class have. You can define methods in that class that all instances of that class will understand.

As one of our basic rules, all computation in Smalltalk proceeds from messages. It shouldn't be surprising that creating classes, instances, and methods is all done from messages, too.

The basic format of the message to create classes looks like this:

```
Object subclass: #NameOfClass
        instanceVariableNames: 'instVarName1 instVarName2'
        classVariableNames: 'ClassVarName1 ClassVarName2'
        poolDictionaries: ''
        category: 'Collections-Abstract'
```

The message is sent to the superclass. In this case, it's already set up to be **Object**. Replace the **NameOfClass** with the name of the class that you want to create, but leave the **#** there. It's a necessary part of the syntax. Replace **instVarName1 instVarName2** with the names of any data variables that you want all instances of the new class to have. You very rarely need class variables, so you can just delete **ClassVarName1**

ClassVarName2—but leave the quotes! (Remember, this is a message, and a string must be passed in as an argument, even if it's an empty string.) Ignore pool dictionaries, too. Finally, you can structure your classes into groups by defining their category.

> ***CautionaryNote:*** Smalltalk *is* case sensitive. **Person** is not the same as **person**. Standard style in Smalltalk is that all classes and global variables are capitalized. All instance and local variables begin with a lowercase letter. Multiple words are combined in Smalltalk using the mixed case notation, such as **NameOfClass** above.

Here is a filled-out message that creates a class called **Person**, where instances of **Person** know their **name** and **address**.

```
Object subclass: #Person
        instanceVariableNames: 'name address'
        classVariableNames: ''
        poolDictionaries: ''
        category: 'People-Project'
```

If you select the above and PrintIt, you will create a new class in your image called Person. To create a new **Person**, just send the message **new** to the class. **fred ← Person new** will create a new **Person** instance and put it in the variable **fred**.

Methods always have the same format:

```
messageForThisMessage
        Smalltalk-statements-to-execute-for-this-message
```

We can define a new method with a message to the class. Because the **compile:** message takes a string, we have to embed quotes in our string if we want them. We do that by duplicating the quotes. The classification string allows us to create groups of methods that have similar functionality. In this case, we'll call this a kind of **Greeting** method.

```
Person compile:
        'greeting
                Transcript show: ''Hello world!'';cr.'
                                classified: 'Greeting'.
```

If we now PrintIt **fred greeting**, we'll get a **Hello world!** in our Transcript from **fred**.

The rule that "Everything is an object" is still true with respect to classes. Classes *are* objects. Unlike object-oriented programming languages like Java and C++, classes in Smalltalk can understand messages that the instances of the class do *not* understand. For example, *new* is understood by classes in Smalltalk, but not by instances of those same classes.

It is also still true that "Everything is an instance of some class." Classes are instances of other classes called *metaclasses*, which, in turn, are subclasses of the class **Metaclass**. But metaclass programming can get pretty complicated, and we won't be getting into it in this book.

All of this said, nobody programs Squeak like this. Squeak provides wonderful tools for programming that require no one to memorize the syntax of message like these. In the following sections, the environment of Squeak becomes the focus.

CautionaryNote: Somewhere along here, when creating new classes and methods, Squeak will ask for your initials. Go ahead and enter them, then press Accept. Squeak labels new code in the changes file with your initials, so that when you share code, it's possible to see who wrote what.

3.4 The Squeak Model of Execution

Squeak doesn't work the way that you may think about programming languages working. In languages like C or Pascal, the mental model of how the language works is simpler. Simple statements (like assignments and if-thens) are executed serially. Control structures like while and for loops are well-defined with reserved functionality: Programmers cannot invent new control structures. There are function calls that can be mapped to either library-based functions or programmer-provided functions.

But a statement like **12 printString** cannot be explained with this kind of model. **printString** is not predefined in the language, and its meaning can be rewritten by the programmer. The mapping from the word **printString** to a piece of code that actually executes is not direct.

Here is a way to think about how Squeak executes statements.

- Arguments are evaluated first, following precedence rules.
- The message and its arguments are sent to the receiving object.
- The class for the receiving object is checked to see if it has an instance method for the given message. If so, the method is executed—following this same model of execution.
- If not, the parent class is checked, and then the parent's parent class, all the way up to the class **Object**.
- If a method is not found for the message, a **doesNotUnderstand**: message (with the original message, an instance of the class **Message**, as an argument) is sent to the original object. Interesting behavior can be created by *overriding* the default behavior of **doesNotUnderstand**:, but the default behavior (in the method in **Object**) is to open an error notifier.
- If execution arrives at a *primitive*,

As complex as this process seems, it's actually quite quick and quite flexible. It predefines very little and allows the programmer maximum flexibility.

Exercises: On Squeak the Language

1. Can you find the implementing method for **whileTrue**:? For integer addition?
2. Almost all statements in Smalltalk are of the form **recieverObject message**. We have seen two syntactic forms in Smalltalk that break that rule. What are they?
3. Write a piece of workspace code, using the language elements of the previous sections to do the following:

(a) Replace all vowels in the string 'Squeak' with dashes.

(b) Compute the average of a set of integers in an array.

4 Using Squeak

The first thing you need to do is to get Squeak itself for your platform. You can get it from the CD included with this book, or from the Squeak website at http://www.squeak.org. Squeak is available for most desktop platforms (and a few palmtop and set-top box platforms). You are going to need four files.

- A *sources* file. This is where all the source code for Squeak is stored. Theoretically, if you could just compile all of the sources, you'd have an *image* file.
- An *image* file. This is the binary (bytecode) of the sources that you will execute.
- A *changes* file. This is where your code that you add to Squeak will go. It's kept separate from the sources file to separate the distribution from what individuals add. The most important thing about the changes file is that it saves *everything* that you write, as soon as you do it. It's automatic backup. If anything goes wrong (and yes, you can crash Squeak), none of your code is ever lost. It's stored, as text, in the changes file.
- An executable *virtual machine* (VM). This is machine dependent and allows your machine to understand the Squeak bytecodes (the machine language of Squeak).

This is what it might look like on a Macintosh when you get all the pieces unpacked. Don't be worried about having *extra* pieces, like a ReadMe file or additional files like *Squeak3D*.

Smalltalk has a different model of programming than you might be used to, in comparison with more traditionally compiled languages such as C or Pascal. There are not separate code files lying around. (Actually, you

can create code files for sharing with others, but they're only useful when you *file them in* for use in your image file.) Instead, you write your programs while executing in Squeak! Squeak is both a language and a complete development environment with editors, debuggers, inspectors, and other tools. As you work, your code gets stored to the changes file, and your binary object code gets added to the image in memory (which you need to save to disk in order to be able to reuse it later.)

Everything that you do goes into the changes file as soon as you do it: Every DoIt, every new class, every new method. This means that if you crash Squeak, your work isn't lost. It's probably in the changes file. The changes file is just a text file -- you can copy out anything that you need to recover from. From the Desktop Menu, you also have access to several changes utilities that let you look over your changes file and recover lost things. From the Desktop Menu, select *Changes*, then *recent change log* to find see all changes from every quit or save that you've executed.

The sources file and the executable remain virtually unchanged when you use Smalltalk. (It possible to save your changes into the sources file, but you rarely really need to.) *The image and changes file, however, need to always be manipulated in pairs.* You can create yourself a new image (by doing *Save As* from the Desktop Menu), you will also create a changes file of the same name at the same time. It makes sense that these two files have to be kept in synch. The image file is the binary executable of the virtual machine. The changes file (with the sources) is the source for that executable.

CautionaryNote: Always keep a "fresh" image available on your disk. Save your image as a new name, and use the new image. That way, if you crash your image, you can always recover into the "fresh" image. When you know that (a) all your text is always saved and (b) you can quickly start over in a new image, you feel much more free to experiment. Later chapters will explain how to recover source code from a "broken" image.

You start the VM with the image file whatever way works for your platform. On Macs and Windows, you can probably just double-click the image file, or else drag it onto the VM file. On UNIX boxes, you'll type a command like **squeak squeak.image**. Soon, you'll see something like this:

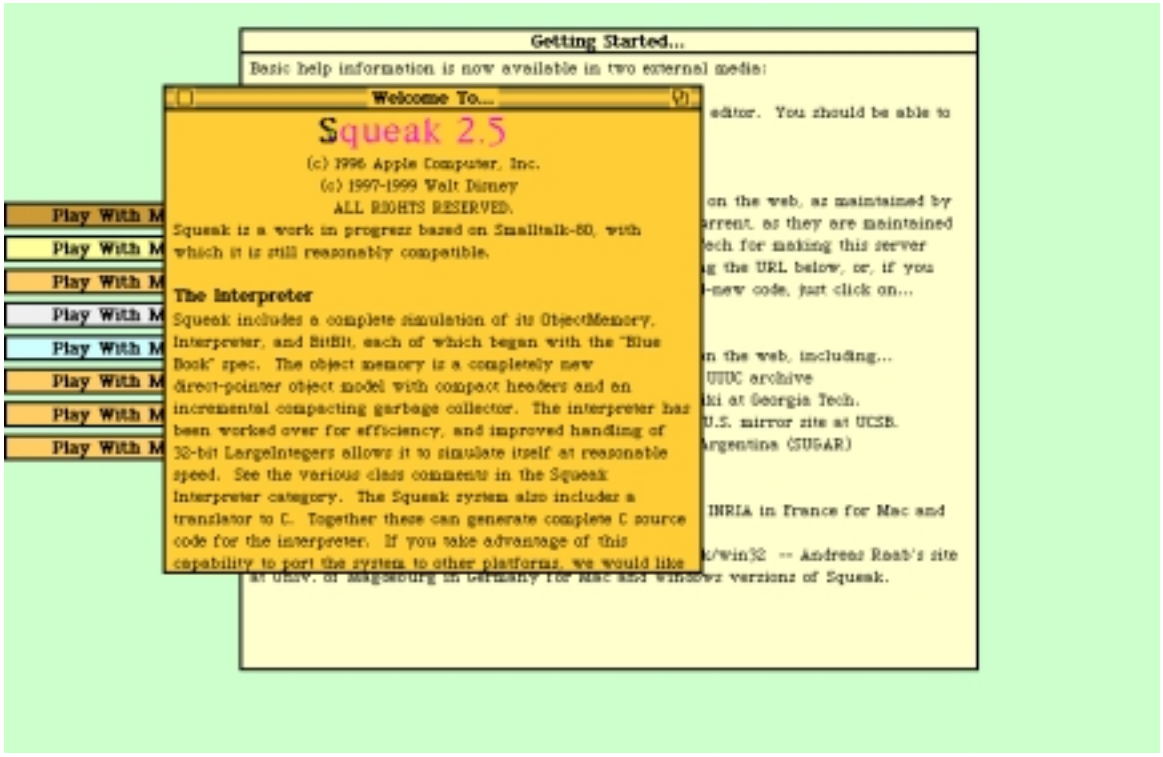

Figure 8: Start-up screen of Squeak 2.5

This is Squeak. Some of the windows look obviously like windows. Others are collapsed windows showing just the title bars. Click the boxes in the right corner of the title bar to toggle expansion of the window. The windows in the initial Squeak display contain interesting demos, information about Squeak, and other neat things. Do play with them at some point.

Click anywhere on the desktop where there is no window, and you will get a menu that looks like Figure 2 at the beginning of this chapter. This is where you create new windows and start activities in Squeak.

A brief tour of the menu items from the Desktop Menu:

- *Keep this menu up* creates a window with the same items, so that they're always available.
- The project items let you jump between projects. Projects have their own set of menus and remember in a special way all the code entered into them, so that you can save out all the code from a single project regardless of how many different classes you worked on.
- If the display becomes messed up from experiments, *restore display* will fix it.
- *Open…* allows you to open a variety of tools and projects.

- *Windows…* provides tools for managing windows, collapsing windows, and re-opening them.
- *Help…* has a grabbag of tools and options. *Update code from server* under the Help menu downloads the latest version of your Squeak from a central server. *Command key help* provides a list of all the special keys available when text-editing, including font selection and options for creating clickable active text. *Preferences* lets you predefine options like always showing scrollbars (instead of the default pop-up scrollbars).
- *Appearance…* lets you predefine things like the color of the windows in Squeak.
- *Do…* provides a set of easily accessible commands to execute. This set of commands is easily user-definable, so that you can create your own "menu items" under this menu.
- *Save* saves the current state of your image into your current image file. *SaveAs* prompts you for a name (e.g., *mySqueak*) to save an image and changes file in. *Save and Quit* saves then quits. *Quit* just ends your session.

4.1 Starting a New Project

You should start working in a new project, without all of these windows cluttering things up, but without having to close any of them and thus losing their contents. Choose *Open…* menu item, and then *Project (MVC)*. A small window appears on your desktop. Click and hold in that window, and you'll get an option to *Enter* the project. Do so.

> ***SideNote:*** You could use *Project (Morphic)*, and all would work well for most readers. MVC is an older interface infrastructure. It works better on older and slower computers. Morphic is the newer interface infrastructure, and it's where the future of Squeak lies. The differences between MVC and Morphic are described more in Chapter 5.

All the windows go away! Actually, they're back in your parent project. Here in this project, you can set up windows to your liking without disturbing the others. You can have as many projects as your memory will allow, and nest them however you like (e.g., all off on the toplevel project, one inside another, whatever). You can always get back out by choosing *Previous Project* from the Desktop Menu. Go ahead and do that. Name your project by clicking on the current name of the window, typing a new name, then hitting return. Re-enter your project.

You do your programming in Squeak in a set of windows that serve as *browsers* and other facilities. We've already seen Transcripts and Workspaces. Let's open a Transcript for displaying text in. Choose *Open…*

then *Transcript* (Figure 5). You can drag around the lower right hand corner of the window to resize it to your liking.

It's easy to write some code that will put something in the Transcript. Choose *Open…* again and open a *Workspace*. The workspace is basically a blank piece of text editor. Type into your workspace:

```
Transcript show: 'Hello, World!'. Transcript cr.
```

Select those lines of text after you typed them. We have been using only the PrintIt option for executing code, but there are several ways of getting the lines to be executed.

- On a Macintosh, type Command-D for "DoIt." On a Windows-based computer, type Alt-D.
- If you have a two or three button mouse, press and hold the second (middle, if you have three buttons) button on your mouse inside the workspace. You'll get a text-editing menu with options to "find" text and such. One of these options is "Do It". Select that. If you have a Macintosh with a one-button mouse, press the Option key as you click your mouse button.

You should see the text **Hello, World!** appear in the Transcript.

You will want to save the state of your session with Smalltalk occasionally, so that you don't lose things in case of a crash. If you go back to the desktop menu, you will see options to **Save** (saves your current image so that all windows and everything else are just as they are when you restart the image), **Save As** (save the image to a new name), and **Save and Quit** (saves the image, and then quits Squeak). If you ever have a crash *before* you save your image, don't worry! Everything you do is always stored in text form to the changes file. From the desktop menu, you can choose changes and you will find a variety of methods for looking through the changes file and recovering things that were lost in a system crash.

CautionaryNote: If you trash your image, you can grab the text of your work out of the old changes file and file it into the new image. There are tools to help you with it, but a good old-fashioned text editor works, too

The changes file has been an absolute necessity for Smalltalk programmers over the years. Everything in Smalltalk is written in Smalltalk, including things like the definition of windows, integers, and other basic building blocks of the system. A programmer can easily do something that makes the image absolutely unusable (say, delete the Integer class). The changes file is what makes sure that work isn't lost even if the image is now trashed.

4.2 Extended Example: Muppets in Squeak

Let's create some classes and a small example as a way into Squeak programming. You will do most of your programming in Squeak within a System Browser. A browser lets you inspect the code that is currently executing within your image. Choose *Open…* and *Browser* from your desktop menu.

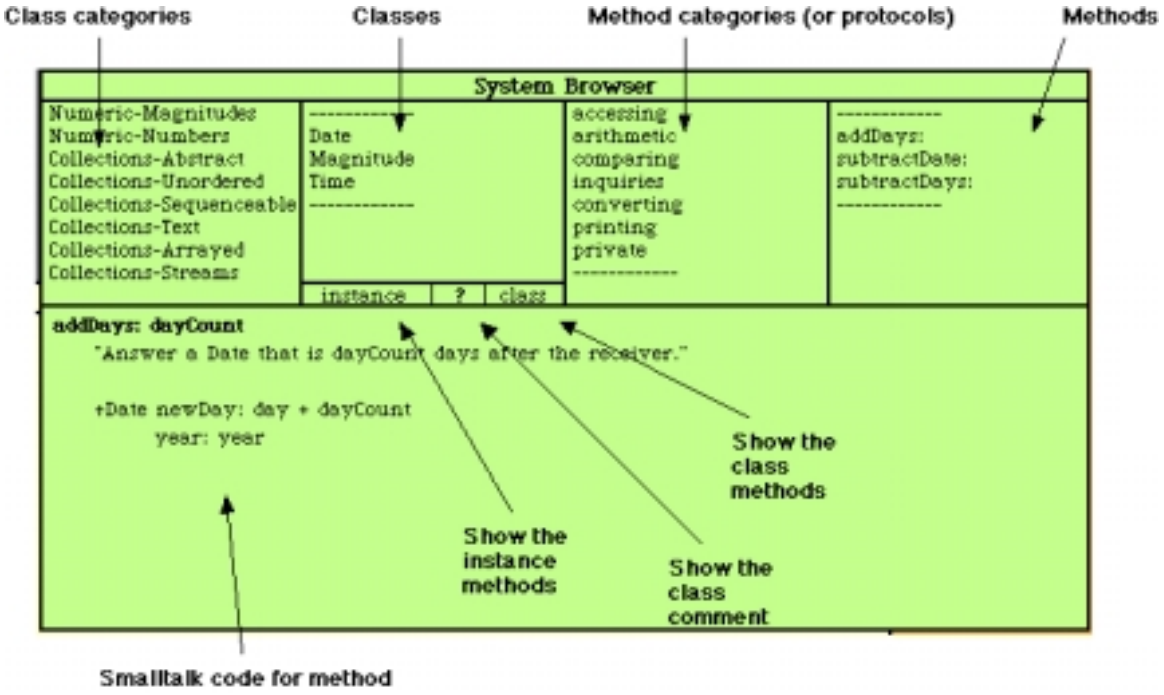

Figure 9: Annotated System Browser

- *Class categories* do not mean anything in terms of the Squeak language. They are just shelves for grouping classes.
- *Classes* are important to Squeak. These are the objects that create instances for you and which serve as the template for your objects. Think of a class as an *object factory*. A class creates objects of a particular kind, with particular factory settings.
- *Method categories* (also called *protocols*) group methods into types: For printing, for accessing data in the object, for iterating, and so on.
- Finally, *methods* are the units for executable Smalltalk code.
- Classes actually serve as the entry point for two different kinds of methods. There are methods that the class itself understands. **new** is a good example of a method that the class itself understands. There are methods that instances of the class understand, such as **greeting** and **do:** The instance/class buttons in the browser allow you to switch between the sets of methods associated with a class. *Almost always* you will want to have the instance button selected.

CautionaryNote: A very common bug in programming practice is to create a class category (say, **Person**) then find that Squeak complains when you try to execute **Person new**. A class category is *not* a class.

Usually start a new programming task in Squeak with a new class category. With your mouse over the class category pane, press your middle mouse button (the right button on a two-button mouse, or option-click on a one-button Macintosh mouse). Over the class category pane, you get the option to create a new class category. Name it something like *Muppet Classes*.

The original Smalltalkers also got confused talking about which mouse button was which, so they came up with a set of position-independent terms for the mouse button. The pointing mouse button is called the "*red*" button, the middle mouse button is called the "*yellow*" button, and the rightmost mouse button is called the "*blue*" button. The red button is always used for pointing, and the yellow button brings up a context-sensitive menu that is dependent on where you're pointing.

Operating System	Red Button	Yellow Button	Blue Button
Macintosh	Mouse click	Option-click	Command-click
Windows	Left-click	Right-click	Alt-left-click
UNIX	Left button	Middle button	Right button

When you select the Muppet Classes category, the browser displays a template for creating a new class (Figure 10). This template is the same message from back in Section 3.3 *Creating Classes, Instances, and Methods.* You don’t ever have to type the message. Simply select a class category, and the template is provided for you to fill in.

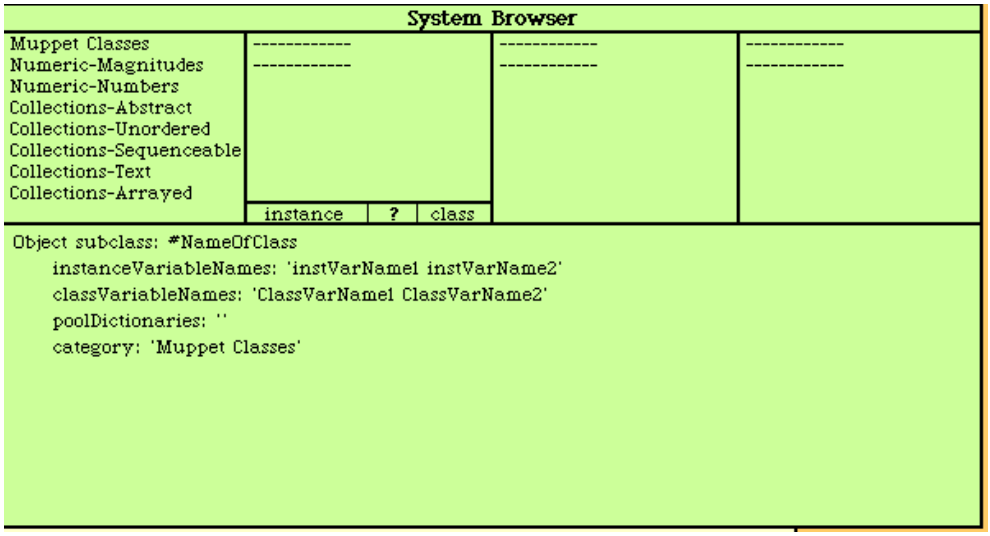

Figure 10: A Browser ready to create a new class

As we saw earlier, you literally just fill in the obvious spots on this template to define the class you want. Leave **Object** as the first word—that's the *superclass*. Object is a very frequent superclass. (As in the basic rules, *every class is a subclass in Smalltalk*.) Change the **NameOfClass** to be **Muppet**. You don't need any class variable names (**classvariablesNames:** in the template), so *delete* everything inside those quotes, but leave the quotes themselves. The *instance variables* are the names for the data that all objects of this type will have. Select everything inside the single quotes on the line **instanceVariableNames**: and type simply **name**. This will let every new Muppet have a name.

To get Squeak to compile the definition you have created, simply choose "Accept" from the text pane where you typed your definition. (Or type Alt-S/Apple-S to accept or *save*.) You will now have a **Muppet** class appear in the class list.

Before we do anything else, we can create Muppet instances, and they know how to do things. Type into your workspace:

```
kermit := Muppet new.
Transcript show: kermit printString
```

Select this and DoIt. The Transcript will now read **a Muppet**, because the default way to print an object is simply to give its class name. Kermit knew how to respond to **printString** because Kermit is an instance of **Muppet**, and **Muppet** is a subclass of **Object**. Everything that **Objects** know, **Muppets** know. **Object** provides a method for **printString**, so Kermit knows how to **printString** .

Now let's actually teach Muppets to do something. Select the default message category **no messages** and a method template appears in the text pane.

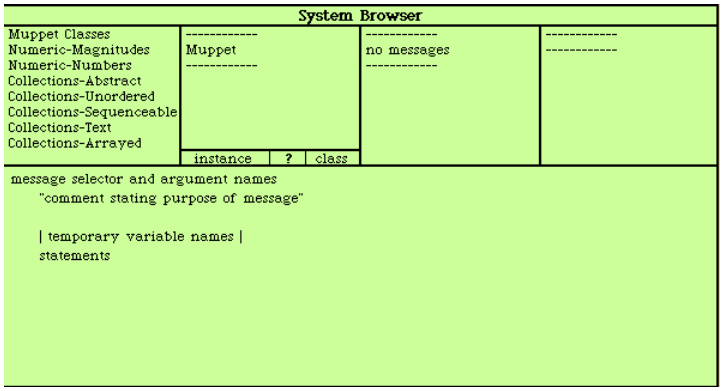

Figure 11: Browser with method template

Now you edit the template to create a method. You edit just as you would any other text in any word processor: Select text you want to change and start typing. We are creating a method to have the **Muppet** greet us in the Transcript. The first method we'll create is to return a general greeting. Don't worry about trying to get the boldface "greeting." Squeak will boldface that for you when you save.

```
greeting

    "Return a pre-defined greeting"

    ^'Hello there!'
```

To get Squeak to compile this method, store the object code into the image, and store the source code into the changes file, all you do is to *accept* the method. Use the context-sensitive menu (remember that it's attached to the *red* button) to choose **Accept**. (On a Mac, command-S will also accept. On Windows, alt-S will accept.) You will see the method name **greeting** appear in the rightmost pane of the System Browser.

The first line of the method is just the message selector, **greeting** . The second line is a comment explaining the method. The third line is the one and only statement in this method. It says to return to whoever sent this message with the String object **'Hello there!'**.

Now, select all that greeting method, and type this in instead.

```
greet

    "Have the Muppet greet us to the Transcript."

    Transcript show: self greeting; cr.
```

Accept this one, too. You now have two methods for **Muppets**. This is the method that will call the greeting method and actually display things

to the Transcript. **self** is a special reference in Smalltalk. It always refers to the object that received the original message which led to this method being executed.

self won't always be an instance of **Muppet**. Let's say that you create a subclass of **Muppet** called **FrogMuppet**. An instance of **FrogMuppet** might have a different greeting from that of a **Muppet**. By sending the message **self greeting**, we ask the instance to give us its greeting. In this way, the subclass's (**FrogMuppet's**) method would *override* the one in the superclass (**Muppet**).

We can now use these methods. Try typing this in the workspace and do it.

```
kermit := Muppet new.
kermit greet.
```

You'll see the greeting appear in the Transcript. Here is exactly what happened in those two lines that were just executed:

1. **Muppet** was asked to create a new instance. It doesn't know how (it has no class methods right now, let alone one for **new**), so it passes the request up to its superclass. **Object** does know how to **new**, so by inheritance, **Muppet** does, too. A new instance of **Muppet** is returned.
2. **kermit** is a variable that is bound to a new instance of **Muppet**.
3. The new instance of **Muppet** is asked to **greet**.
4. The **greet** method will send out to the Transcript whatever **self greeting** returns.
5. The message **greeting** is sent to **self**, and it returns '**Hello there!**'.
6. The **cr** message puts a carriage return to the Transcript

We need some ability to set the name of the Muppet, if we ever want to use the Muppet's name. No internal data of an object can be manipulated directly. If we want any method external to Muppet or piece of code in a workspace to set the name of a Muppet, we must have a method to do it, Select all that text again, and type this one:

```
name: aString
    "Set the name of the Muppet"
    name := aString.
```

We could also define a method to allow an external object query the value of an object's name. That method would look like this:

```
name
```

```
"Get the name of the Muppet"

^name
```

Smalltalk has no problem distinguishing between **name**: (which takes an argument) and **name** (which does not). A colon is a significant character in the name of the method. It indicates where arguments appear in the message: A colon appears at the end of each keyword that precedes an argument.

Now, let's redefine **greet** so that it presents the name, too. Reselect the greet method so that it's showing in the Browser, enter the below (just add the last line), then re-accept. A new definition of **greet** will then be entered into the system.

greet

```
"Have the Muppet greet us to the Transcript."

Transcript show: self greeting; cr.

Transcript show: 'My name is ', name; cr.
```

Accept this last one. Now you have enough to have a fully functioning Muppet! In a workspace, type this:

```
| someMuppet |
someMuppet := Muppet new.
someMuppet name: 'Elmo'.
someMuppet greet.
```

If you don't have a Transcript open, open a new one. Then select all the code in your workspace and choose **DoIt** from the yellow button menu (or type Command-D on a Mac, or Alt-D on Windows). You should see Elmo introduce himself and greet you in the Transcript. (You could have also used PrintIt, but we don't care about the return value from this workspace code.)

There are several ways to save your work in Smalltalk.

- You should frequently save your image from the Desktop Menu. That writes out a new images file.
- You can also *fileOut* your code. A fileOut is a text-only representation of your code. It can filed back in from the **File List**, which can be opened from the *Open…* menu (use the yellow button menu on a filename). Wander through the System Browser to try the yellow button menu over various panes. You'll find that you can fileOut a whole class category, just a class, just a single method category of a class, or even a single method. The best reason for doing your work inside of a project is that you can `fileOut` all the changes made within a project, in whatever class or category—take a look at a *Simple Change Sorter* or *Dual Change Sorter* from the *Changes…* menu.

> ***HistoricalNote:*** It's quite appropriate for the first example of using Squeak to involve the Muppets. The Xerox PARC *Alto* was developed to be the "interim Dynabook"—a place to explore Dynabook ideas until the hardware could catch up. The very first test of the Alto was to move the image of Cookie Monster across the cool new bitmap display.

Exercise: On Muppets

4. If Kermit was actually an instance of **FriendlyMuppet** (a subclass of **Muppet**) whose greeting returned 'Well, Howdy!', how would the above chain of events change? Create **FriendlyMuppet** as a subclass of **Muppet**, create a new greeting method, and try the above example with Kermit as a **FriendlyMuppet**.

5. Not all Muppets greet you with "Hello there!" Kermit, being an especially friendly Muppet, would say "Hey-Ho!" Oscar, being an especially grouchy Muppet, would say "Go Away!" Create the subclasses **FriendlyMuppet** and **GrouchyMuppet** with Muppet as the superclass. By adding a **greeting** method in each (thus *overriding* the one in Muppet), we can specialize the greeting for each kind of Muppet.

6. Our method category name has been turned into "As Yet Unclassified" instead of "No messages," but that isn't very clear. You can select the name and change it using the context-sensitive menus. There may not be a single name that classifies all three of these methods. Both **greet** and **greeting** are about "Greeting," but **name:** is about "Accessing" (data). If you create new categories, you can reorganize methods to make sense. Use the YellowButton menus in the message protocols pane to add new categories. Create at least the Accessing method category. Use the *Reorganize* menu item to change where the messages go. When you do, you get a list of methods and categories in the code pane. Copy paste until the methods are in the methods

```
('accessing')
('as yet unclassified' greet greeting name:)
```

7. We didn't implement the message **new** for **Muppet**. Where is the method that is processing it?

8. One of the control structures that basic Squeak is missing is some kind of *switch* or *case* statement. Build one that can be used like this:

```
#(      ('a' [Transcript show: 'An a was input'])
        ('b' [Transcript show: 'A b was input']))
        switchOn: 'a'
```

In this example, if the input is 'a', then the first block is executed. If it's 'b', then the second block. Note that the parentheses inside the #() will define sub-arrays.

5 Generating: Finding More in Squeak

That's your whole introduction to using Squeak! There are lots of external resources to help you in learning to Squeak, like the Squeak

Swiki at http://minnow.cc.gatech.edu/squeak.1 and the Squeak Documentation website at http://minnow.cc.gatech.edu/squeakDoc.1

But there is also a great deal of internal support within Squeak to find things out. There aren't big books of API (Application Programmer Interface) calls for Squeak. First, they would do little good because you change them all the time—here's nothing hidden or unmodifiable in Squeak. But more importantly, it's pretty easy to find anything you want to find in Squeak.

Here is a collection of sample questions or situations that Squeak can help you with. These are useful to demonstrate several of Squeak's tools for helping programmers find things.

5.1 Finding Classes: There has got to be a Window or a TextField class around here somewhere. Where is it?

There are several ways of trying to find classes like that.

- If you know the name of the class, type it anywhere (say, in a Workspace window), double click on it to select it, then choose *Browse* from the yellow button menu (Command-B Macs, Alt-B Windows). A System Browser will open with the right class selected.
- If you have a System Browser open, you can do a *Find Class* from the yellow button menu over the class categories list. Type the name of the class (or even a portion of a name, like "window"), and you'll get a list of names to choose from.

But what if you have no idea what the name of the class is? There are two strategies. First, walk down all the categories in the class category list. There isn't that many, so it won't take you too long, but it will give you a sense of the kinds of classes located under each kind of category.

The second strategy is to find an instance of the kind of thing you want and *inspect* it. Every object understands the message **inspect**. If you can write a Smalltalk expression that is the kind of thing that you want, then you can inspect it. Try typing **Transcript** in any window, select it, then choose *Inspect* from the yellow button menu (also Command-I on Macs, Alt-I on Windows). Equivalently, you might also DoIt on **Transcript inspect**.

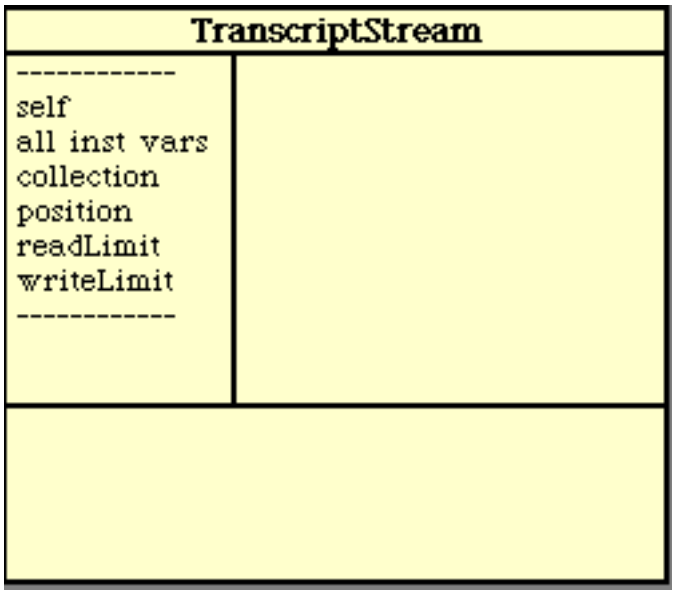

Figure 12: An inspector window

You get an inspector window displaying the class name of the instance (**Transcripts** are an instance of **TranscriptStream**) and displaying all the instance variables of the instance. You can click on each of the instance variables to see its value. You can also ask to inspect any one of these using the yellow button menu. You may also choose *browse class* to see the class of the data in the instance variable. (Remember, *everything* is an object, so even data in instance variables are objects, so they have classes, too.)

If you did the above example and are now inspecting Transcript, you can choose **self** and choose *browse class* from the yellow button menu. You'll get another kind of browser from the System Browser seen earlier. The class browser works the same way as the System Browser: For example, you can add new methods from here. From there, you can see how Transcript is implemented and dig further.

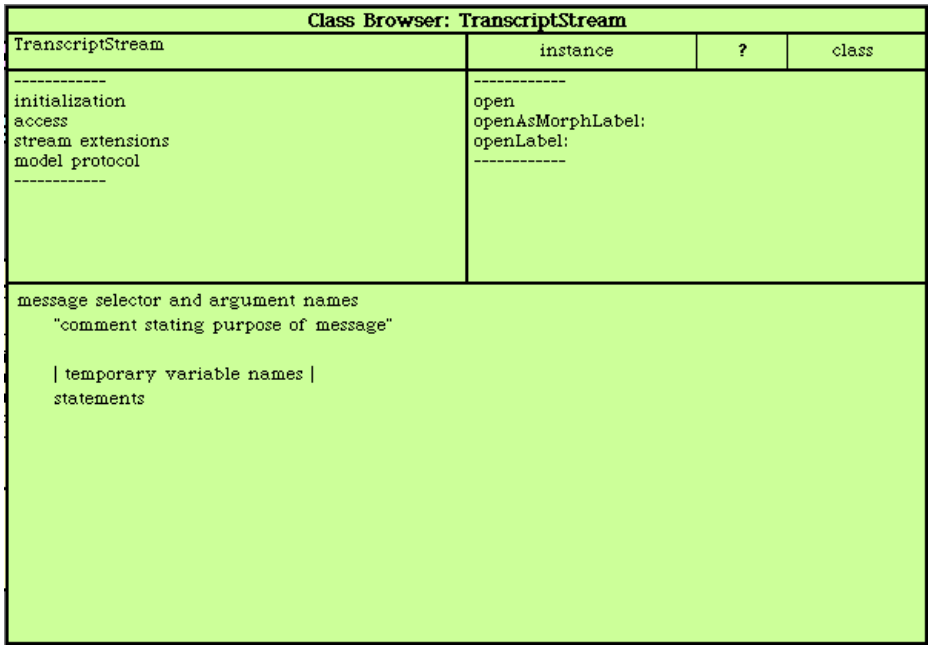

Figure 13: A Class Browser on Transcript's Class

Inspectors are *amazingly* powerful tools when debugging. Play with some of the yellow-button menus in the inspector. As an example of a powerful tool, you can find all references to the object you're inspecting, which can be very useful when trying to figure out why an object hasn't been garbage collected yet. Also, the bottom pane of the inspector is actually a workspace where **self** means the object being inspected! You can send messages to the object while debugging, like **self printString**, by typing them into the bottom pane and doing PrintIt.

5.2 Exploring Objects: I'm exploring a complicated object, and now I've got a bazillion inspectors all over the screen. Is there some other way to explore an object?

In Squeak 2.5, a new kind of inspector was introduced called the Object Explorer (by Bob Arning, one of the Open Source contributors to Squeak.) Instead of **inspect**, send the message **explore** to any object. The Object Explorer provides an outline view on any new object.

For example, if you wanted to explore how a literal array object is parsed, you might DoIt:

```
#(123 'a' (a b c) 34.5) explore
```

The result (seen in Figure 14) is an outline on the original object. Selecting (clicking) any object allows the user to bring up a yellow button menu that allows the user to open a traditional inspector or a new explorer on the selected item. A traditional inspector allows you to (as above) find the class of the object and browse that class.

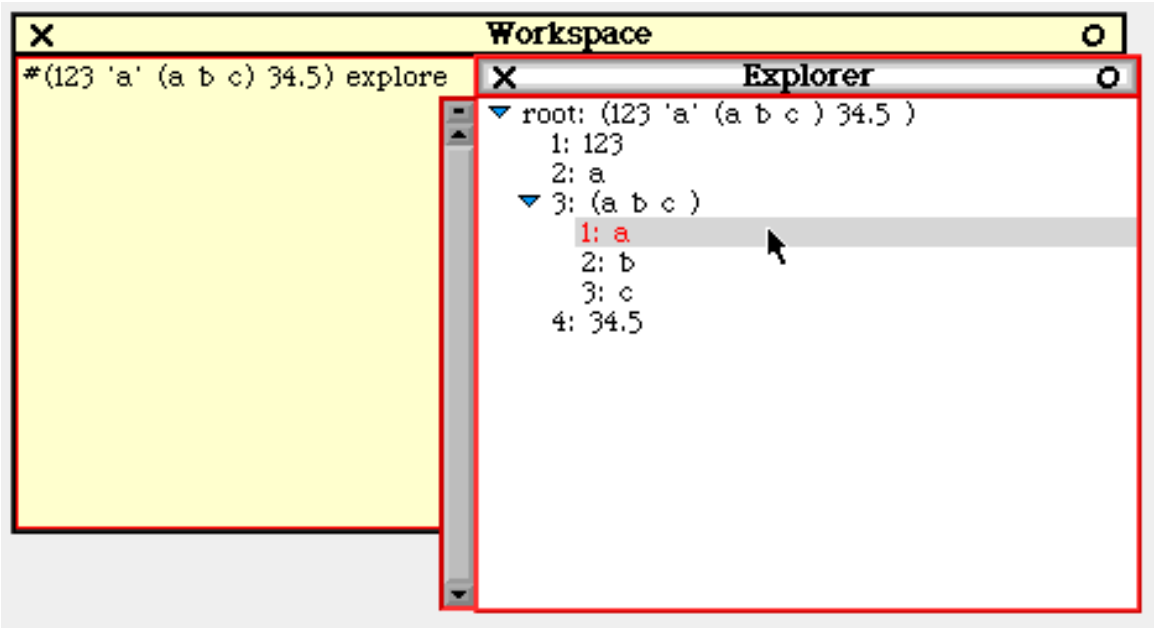

Figure 14: Object Explorer on a Literal Array

5.3 Finding Methods: I remember there's a way of getting an element, 'at'-something, but I can't find it.

Squeak contains a very powerful tool for finding methods in the system. It's called a *Selector Finder*. You can open one from the *Open…* menu on the main Desktop Menu. The bottom pane gives the instructions for its use.

If you remember part of a method's name, but not the whole thing, just type the part you remember in the upper pane, then choose accept (Command-S Macs, Alt-S Windows). The list on the left shows all method names (also called *selectors)* that contain the accepted name. Selecting one of those shows all the specific classes that implement that selector on the right. Choose one of the items in the right list to open up a browser on that method.

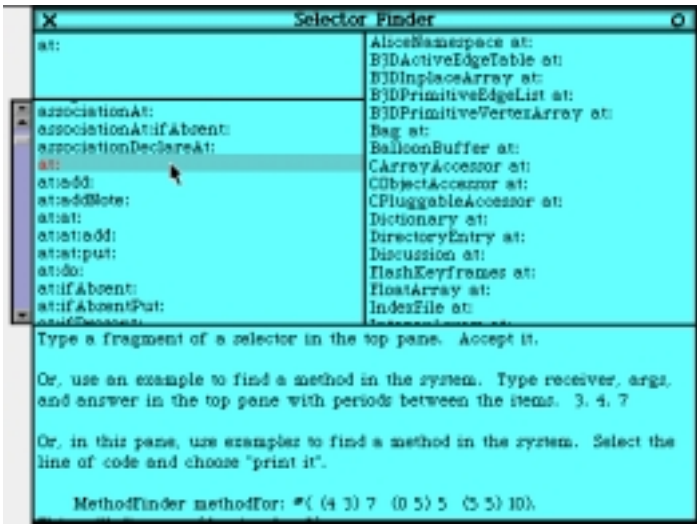

Figure 15: Using the Selector Finder with Part of a Message

Even more powerful is the find-by-example aspect of the Selector Finder. If you know that a method must exist for a set of inputs and an output, you can use the Selector Finder to find the method. You simply type the inputs and the output, separated by periods, into the top pane. Accept, and the selector list will show all those messages that will actually do the operation you describe! **'abc' . 2 . $b** as the input will find the **at:** message (Figure 16).

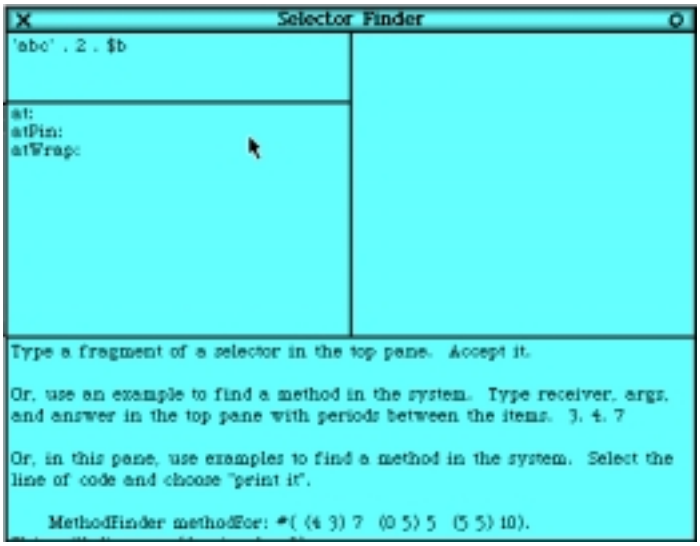

Figure 16: Using the Selector Finder with find-by-example

5.4 Finding Implementors: I see a reference to a method named `findTokens:` (or whatever). Where is it? What does it do?

The easiest way to figure out what anything is, from methods, to class names, to globals, to even constructs like `$c` is to select it and choose

explain from the yellow button menu (you have to choose *more* at the bottom of the first menu.) What you will get is a text description of the thing you selected and a piece of executable Smalltalk code that you can DoIt to open a browser for more information. The text is inserted right after the item you had highlighted.

The information that you get for explaining **findTokens**: is:

```
 "findTokens: is a message selector which is defined in these classes
(String )."
Smalltalk browseAllImplementorsOf: #findTokens:
```

The *explain* option leaves the text highlighted, so that you can just hit Delete to get rid of the text, or DoIt to execute the command that opens a browser.

The next easiest way is to simply select the method and choose *Implementors...* from the yellow button menu (Command-M on Macs, Alt-M on Windows).

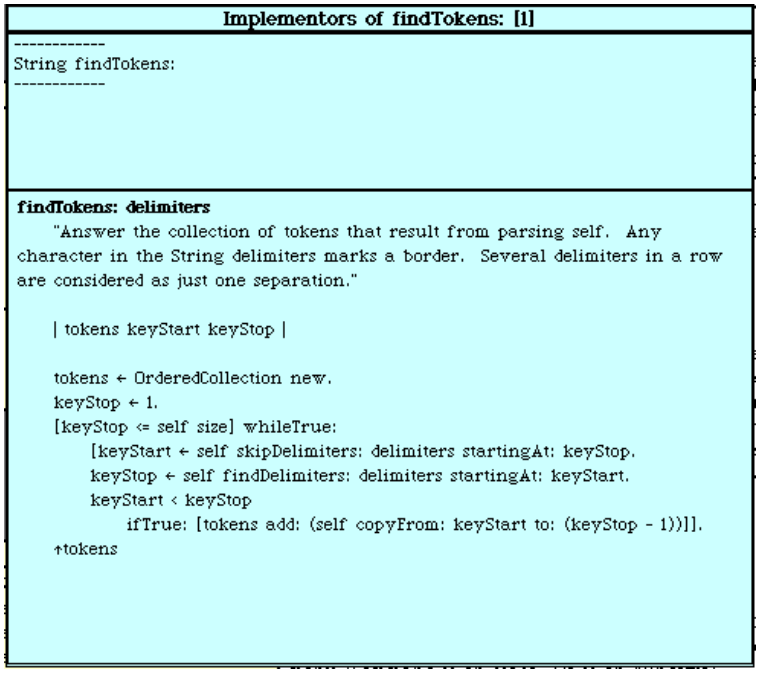

Figure 17: An Implementors browser for findTokens:

We can see here that there is only one class that implements a method for the message **findTokens**:, and that is **String**. We can see the comment for the code and the actual code here.

It should be noted that anywhere that you can see a method, you can edit a method. If you wanted to change **findTokens**:, you could simply edit it from this window and accept. All browsers work as well as any browser.

5.5 Finding Senders: That's what `findTokens:` does. Who uses it?

Select the selector message anywhere that it appears and use the yellow button menu *Senders...* (Command-N on Macs, Alt-N on Windows). It turns out that **findTokens**: is a very popular method that is used frequently to break strings into arrays of tokens.

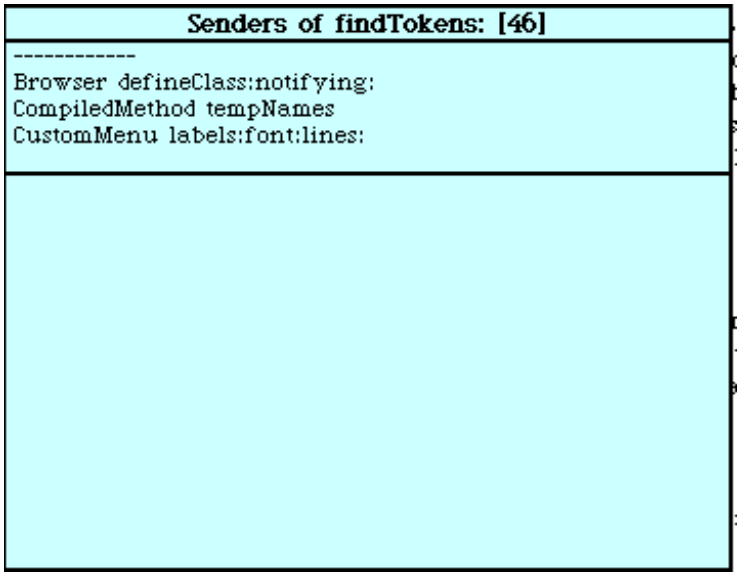

Figure 18: A Senders browser for findTokens:

5.6 Debugging: I can't figure out what my code is doing! I've got an error, but it makes no sense to me.

Squeak errors are often hard to figure out because the error message expects that you understand the basic notions of Squeak. Fortunately, the debugging tools in Squeak are excellent.

Let's say that you executed something like **Transcript show: 34.2**. You will get a notifier like the one (in Morphic–the MVC version doesn't have proceed and debug in the title bar).

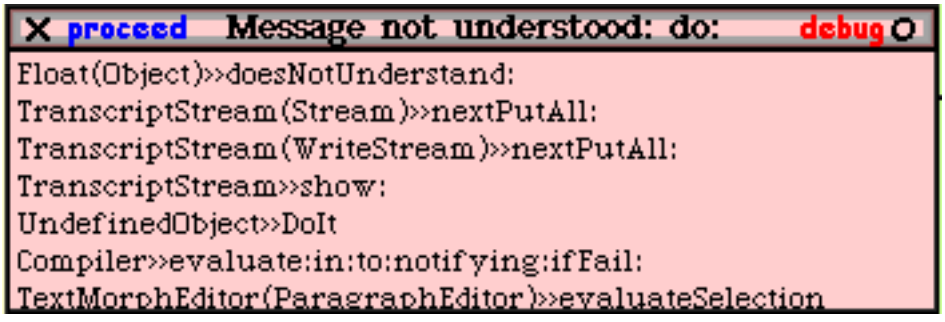

Figure 19: An Error Notifier (in Morphic)

This can be a confusing message because you don't see that you sent the message **do:** anywhere. But remember that your basic message send, **show:**, led to many other message sends (see Section 3.4). It makes sense if you dig deeper. **show:** is meant only to take strings, and you handed it a floating point number.

We can see this directly. Choose *debug* from the notifier's title bar, or use the Yellow Button Menu from within the notifier to choose proceed.

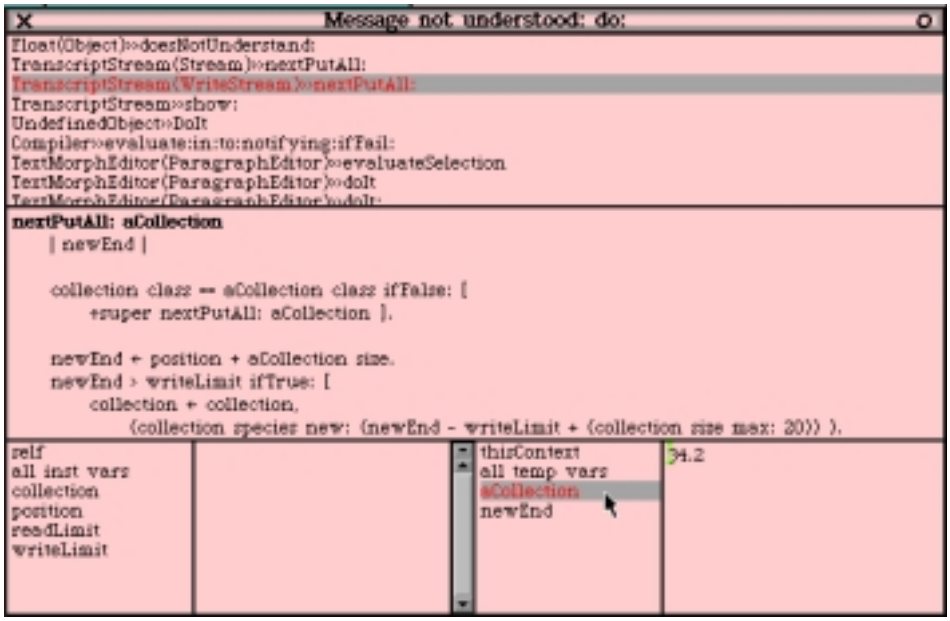

Figure 20: Squeak's Debugger

The top list pane is showing you the backtrace of all methods currently in execution when the error occurred. If you select one of those, you will actually see the method in the middle pane, with the currently executing message selected. The bottom sets of panes are actually inspectors. The left two lower panes are showing the inspector on the object receiving the message. The right two lower panes are showing the inspector on the context of the method, i.e., the local variables and the arguments to the method.

The top message in the list is actually the message that generated the debugger window, so that's never the source of the error. Instead, that's where this debugger window was generated. (Recall, all of Squeak is written in Squeak.)

In this particular example, we can see what happened pretty easily. Five messages down is the DoIt that started this whole process. We can see (fourth message down) that the Transcript did try to do the **show:** message. But take a look at the third message down, the **nextPutAll**:. Selecting that shows the problem. **nextPutAll**: expects a **Collection**, **aCollection**, as an argument. In the lower right panes, we can actually

look at the variables defined in the context of this method. **aCollection** is the argument passed in to **show:** It's not surprising that when **34.2** was asked to **do**:, it didn't know how.

If you open the yellow button menu in the top pane, you'll find that you can **step** (next line within the same method, *whichever* method is currently selected), **send** (following a call into a lower message), **proceed** (go through the code full speed outside the debugger), and other options for executing the code slowly. Again, the code in the middle pane is like code in any browser: You can actually change code and recompile during debugging, then continue stepping through the code after you make a correction.

If you are having trouble tracking your code, you can insert **self halt** anywhere in your code to force an error notifier and thus allow you into the debugger. Once in the debugger, you can step through your code, using the inspectors to check the values of things as you go.

5.7 Learning to Use a Class: What all does `String` or other classes understand?

There are a couple of ways to look at what a class knows how to do. From a System Browser with String selected, open the yellow button menu over the class list pane. You'll see an option to *spawn protocol*. This menu item opens up a browser that shows all the messages that String understands from all of its inherited classes.

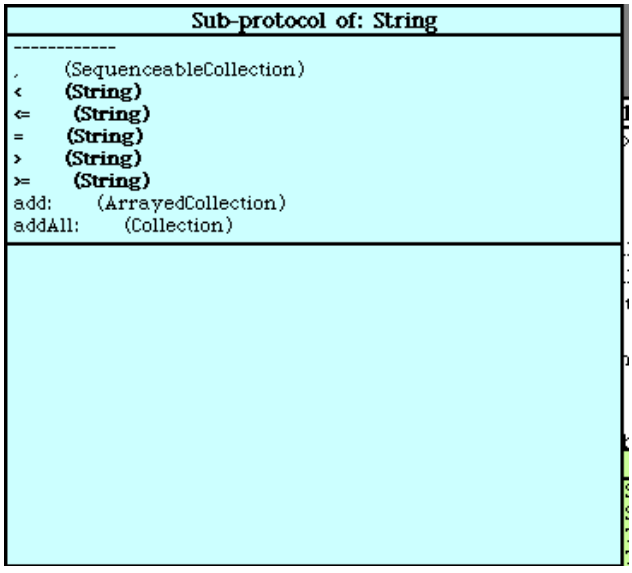

Figure 21: A Protocol Browser on String

For classes that have lots of parents and lots of methods, the protocol list can be too long and intimidating. But it is probably the best browser

for seeing *everything* that a class understands. Another useful option from the basic system browser is to choose *spawn hierarchy*.

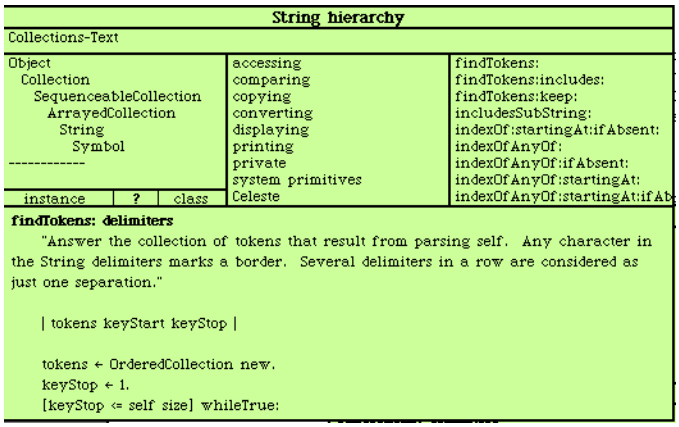

Figure 22: A Hierarchy Browser on String

The hierarchy browser shows you only the classes that are parent (and children, if any) classes of the selected class. You can wander up and down the hierarchy to look at all the methods that the selected class understands.

6 How Do You Make an Application in Squeak?

A common question from students at this point is, "How do I make applications in Squeak? In other languages, I can create an executable file and hand it to them. How do I do that in Squeak?"

It's possible to create something like that in Squeak. Essentially, you can hand someone a stripped down image with only necessary classes in it. Of course, you always have to give people the virtual machine (VM)—that's the only part that's really executable on a native platform. You can get Squeak to do without the sources and changes files. (Hint: Check out *Preferences* under the *Help* menu.) You don't really need sources and changes if the user is not going to be compiling anything. If you set up your image with just your windows available and strip out everything else, you essentially have an "executable" with a VM "runtime".

There's another answer that's more powerful, though. Squeak is based on a version of Smalltalk that is nearly pre-applications. It came before our current software market was established. Of course, people bought and sold applications software then, but what didn't exist was Word, WordPerfect, AutoCAD, and the thousands of individual programs that you buy and sell and lose on your hard disk and have trouble uninstalling.

In Smalltalk, there is an idea of a "Goodie." A Goodie was a piece of code that you would *fileIn* (which we will see in the next chapter) that would give you new capabilities—maybe a new image editor, or a spreadsheet. These had many of the characteristics of applications, yet

were different. When you loaded in the new image editor, all those new classes and methods were also available for other application goodies. Sometimes goodies conflicted. There were tools for figuring that out, and there was always the solution of having multiple image-and-changes sets on your disk. But the interesting thing about goodies that was different than applications was that a "Goodie" was just Smalltalk code. You could look at it and change it. You could use all the powerful features of Smalltalk with it (such as multiprocessing, which we see in a future chapter). The notion of an "integrated suite" was completely transparent—how much more integrated can you get than to have everything as an object in the same image?

Having early Smalltalk running on modern machines with modern extensions allows us to re-think some of the accepted notions of computing that have become entrenched in the last twenty-plus years. Applications and windows used to be software that anyone could change and that could naturally interact. It's an important lesson to consider what strengths that model had and how we might gain those today.

Exercises: Using Squeak

9. Build a binary tree representation in Squeak so that you can create and manipulate tree elements, and do traversals. Implement **inorder**, **preorder**, and **postorder** traversals.

```
root := Tree new.        "Make a new tree"
root left: (Tree new). "Make a left tree"
root info: 'This is the root info'.
root left info: 'The is the left subtree info'.
root left left: Tree new.
root left right: Tree new.
root left left info: 'Left sub-subtree.'.
root left right info: 'Right side of Left sub-subtree.'.
root right: (Tree new). "Make a right tree."
root right info: 'I am the right tree.'.
root inorder  "Return the inorder traversal."
OrderedCollection ('Left sub-subtree.' 'The is the left subtree info'
'Right side of Left sub-subtree.' 'This is the root info' 'I am the
right tree.' )
```

10. Create a tiny painting program. Simply follow the Sensor and move a Pen to follow it. Put up a couple of graphic images that the user might click to change colors or to quit drawing.

11. More Challenging: Create a tiny *drawing* program. Let the user create graphical *objects*: Squares, circles, polygons. Recognize that clicking on the object is different than creating a new object. Let the user drag the objects around. (This will become easier after learning the methods of Chapter 5.)

References

There is a wonderful quick reference to Squeak by Andrew Greenberg at http://www.gate.net/~werdna/squeak-qref.html

The book *Smalltalk-80: The Language* by Adele Golberge and Dave Robson is clearly the best definition of the language that underlies Squeak.