A Tutorial for GNU Smalltalk

Andrew Valencia

Valencia Consulting

Overview

What this manual presents

This document provides a tutorial introduction to the Smalltalk language in general, and the GNU Smalltalk implementation in particular. It does not provide exhaustive coverage of every feature of the language and its libraries; instead, it attempts to introduce a critical mass of ideas and techniques to get the Smalltalk novice moving in the right direction.

Who this manual is written for

This manual assumes that the reader is acquainted with the basics of computer science, and has reasonable proficiency with a procedural language such as *C*. It also assumes that the reader is already familiar with the usual janitorial tasks associated with programming—editing, moving files, and so forth.

1. Getting started

1.1. Starting up Smalltalk

Assuming that GNU Smalltalk has been installed on your system, starting it is as simple as:

```
% mst
```

the system loads in Smalltalk, and displays a startup banner like:

```
Smalltalk 1.1.1 Ready

st>
```

You are now ready to try your hand at Smalltalk! By the way, when you're ready to quit, you exit Smalltalk by typing control-D on an empty line.

1.2. Saying hello

An initial exercise is to make Smalltalk say "hello" to you. Type in the following line ("printNl" is a upper case N and a lower case L):

```
'Hello, world' printNl !
```

The system then prints back 'Hello, world' to you.[1]

1.3. What actually happened

The front-line Smalltalk interpreter gathers all text until a '!' character and executes it. So the actual Smalltalk code executed was:

---

[1] It also prints out a lot of statistics. Ignore these; they provide information on the performance of the underlying Smalltalk engine. You can inhibit them by starting Smalltalk as:

```
% mst -q
```

```
'Hello, world' printNl
```

This code does two things. First, it creates an object of type "String" which contains the characters "Hello, world". Second, it sends the message named "printNl" to the object. When the object is done processing the message, the code is done and we get our prompt back.

You'll notice that we didn't say anything about printing the string, even though that's in fact what happened. This was very much on purpose—the code we typed in *doesn't know anything about printing strings.* It knew how to get a string object, and it knew how to send a message to that object. That's the end of the story for the code we wrote. But for fun, let's take a look at what happened when the string object received the "printNl" message.

1.4. What a string does with a "printNl" message

The string object containing "Hello, world" was sent the message "printNl". It then goes to a table[2] which lists the messages which strings can receive, and what code to execute. It finds that there is indeed an entry for "printNl" and runs this code. This code then walks through its characters, printing each of them out to the terminal.[3] The central point is that an object is entirely self-contained; only the object knew how to print itself out. When we want an object to print out, we ask the object itself to do the printing.

1.5. Doing math

A similar piece of code prints numbers:

```
1234 printNl !
```

Notice how we used the same message, but have sent it to a new type of object—an integer (from class "Integer"). The way in which an integer is printed is much different from the way a string is printed on the inside, but because we are just sending a message, we do not have to be aware of this. We tell it to "printNl", and it prints itself out.

As a *user* of an object, we can thus usually send a particular message and expect basically the same kind of behavior, regardless of object's internal structure (for instance, we have seen that sending "printNl" to an object makes the object print itself). In later chapters we will see a wide range of types of objects. Yet all of them can be printed out the same way—with "printNl".

White space is ignored, except as it separates words. This example could also have looked like:

```
                1234
    printNl       !
```

An integer can be sent a number of messages in addition to just printing itself. An important set of messages for integers are the ones which do math:

```
(9 + 7) printNl !
```

Answers (correctly!) the value 16. The way that it does this, however, is a significant departure from a procedural language.

1.6. Math in Smalltalk

In this case, what happened was that the object "9" (an Integer), received a "+" message with an argument of "7" (also an Integer). The "+" message for integers then caused Smalltalk to create a new object "16" and return it as the resultant object. This "16" object was then given the "printNl" message, and printed "16" on the terminal.

---

[2] Which table? This is determined by the type of the object. An object has a type, known as the *class* to which it belongs. Each class has a table of methods. For the object we created, it is known as a member of the "String" class. So we go to the table associated with the String class.

[3] Actually, the message "printNl" was inherited from Object. It sent a "print" message, also inherited by Object, which then sent "printOn:" to the object, specifying that it print to "stdout". The String class then prints its characters to the standard output.

Thus, math is not a special case in Smalltalk; it is done exactly like everything else—by creating objects, and sending them messages. This may seem odd to the Smalltalk novice, but this regularity turns out to be quite a boon—once you've mastered just a few paradigms, all of the language "falls into place." Before you go on to the next chapter, make sure you try math involving "*" (multiplication), "-" (subtraction), and "/" (division) also. These examples should get you started:

```
(8 * (4 / 2)) printNl !
(8 - (4 + 1)) printNl !
(5 + 4) printNl !
(2/3 + 7) printNl !
(2 + 3 * 4) printNl !
(2 + (3 * 4)) printNl !
```

2. Using some of the Smalltalk classes

This chapter has examples which need a place to hold the objects they create. The following line creates such a place; for now, treat it as magic. At the end of the chapter we will revisit it with an explanation. Type in:

```
Smalltalk at: #x put: 0 !
```

Now let's create some new objects.

2.1. An array in Smalltalk

An array in Smalltalk is similar to an array in any other language, although the syntax may seem peculiar at first. To create an array with room for 20 elements, do:

```
x := Array new: 20 !
```

The "Array new: 20" creates the array; the "x :=" part connects the name "x" with the object. Until you assign something else to "x", you can refer to this array by the name "x".

Changing elements of the array is *not* done using the ":=" operator; this operator is used only to bind names to objects. In fact, you never modify data structures; instead, you send a message to the object, and it will modify itself. For instance:

```
(x at: 1) printNl !
```

which prints:

```
nil
```

The slots of an array are initially set to "nothing" (which Smalltalk calls "nil"). Let's set the first slot to the number 99:

```
x at: 1 put: 99 !
```

and now make sure the 99 is actually there:

```
(x at: 1) printNl !
```

which then prints out:

```
99
```

These examples show how to manipulate an array. They also show the standard way in which messages are passed arguments. In most cases, if a message takes an argument, its name will end with ":".[4] So when we said "x at: 1" we were sending a message to whatever object was currently bound to "x" with an argument of 1. For an array, this results in the first slot of the array being returned.

The second operation, "x at: 1 put: 99" is a message with *two* arguments. It tells the array to place the second argument (99) in the slot specified by the first (1). Thus, when we re-examine the first slot, it does indeed now contain 99.

There is a shorthand for describing the messages you send to objects. You just run the message names together. So we would say that our array accepts both the "at:" and "at:put:" messages.

There is quite a bit of sanity checking built into an array. The request

```
6 at: 1
```

fails with an error; 6 is an integer, and can't be indexed. Further,

```
x at: 21
```

fails with an error, because the array we created only has room for 20 objects.[5] Finally, note that the object stored in an array is just like any other object, so we can do things like:

---

[4] Alert readers will remember that the math examples of the previous chapter deviated from this.

[5] As of release 1.1, GNU Smalltalk does not actually catch this error.

```
((x at: 1) + 1) printNl !
```

which (assuming you've been typing in the examples) will print 100.

2.2. A set in Smalltalk

We're done with the array we've been using, so we'll assign something new to our "x" variable. Note that we don't need to do anything special about the old array—the fact that nobody is using it any more will be automatically detected, and the memory reclaimed.[6] So, to get our new object, simply do:

```
x := Set new !
```

Which creates an empty set. To view its contents, do:

```
x printNl !
```

the kind of object is printed out (i.e., Set), and then the members are listed within parenthesis. Since it's empty, we see:

```
Set ()
```

Now let's toss some stuff into it. We'll add the numbers 5 and 7, plus the string 'foo'. We *could* type:

```
x add: 5 !
x add: 7 !
x add: 'foo' !
```

But let's save a little typing by using a Smalltalk shorthand:

```
x add: 5; add: 7; add: 'foo' !
```

This line does exactly what the previous example's three lines did. The trick is that the semicolon operator causes the message to be sent to the same object as the last message sent. So saying "; add: 7" is the same as saying "x add: 7", because "x" was the last thing a message was sent to. This may not seem like such a big savings, but compare the ease when your variable is named "aVeryLongVariableName" instead of just "x"! We'll revisit some other occasions where ";" saves you trouble, but for now let's continue with our set. Type either version of the example, and make sure that we've added 5, 7, and "foo":

```
x printNl !
```

we'll see that it now contains our data:

```
Set (5 'foo' 7)
```

What if we add something twice? No problem—it just stays in the set. So a set is like a big checklist—either it's in there, or it isn't. To wit:

```
x add:5; add: 5; add: 5; add: 5 !
x printNl !
```

We've added "5" several times, but when we printed our set back out, we just see:

```
Set (5 'foo' 7)
```

What you put into a set with "add:", you can take out with "remove:". Try:

```
x remove: 5 !
x printNl !
```

The set now prints as:

```
Set ('foo' 7)
```

The "5" is indeed gone from the set.

[6] This is known as "garbage collection." It is generally done when Smalltalk finds that it is running low on memory.

We'll finish up with one more of the many things you can do with a set— checking for membership. Try:

```
(x includes: 7) printNl !
(x includes: 5) printNl !
```

From which we see that x does indeed contain 7, but not 5. Notice that the answer is printed as "true" or "false". Once again, the thing returned is an object—in this case, an object known as a boolean. We'll look at the use of booleans later, but for now we'll just say that booleans are nothing more than objects which can only either be true or false—nothing else. So they're very useful for answers to yes or no questions, like the ones we just posed. Let's take a look at just one more kind of data structure:

2.3. Dictionaries

A dictionary is a special kind of collection. With a regular array, you must index it with integers. With dictionaries, you can index it with *any* object at all. Dictionaries thus provide a very powerful way of correlating one piece of information to another. Their only downside is that they are somewhat less efficient than simple arrays. Try the following:

```
x := Dictionary new.
x at: 'One' put: 1 !
x at: 'Two' put: 2 !
x at: 1 put: 'One' !
x at: 2 put: 'Two' !
```

This fills our dictionary in with some data. The data is actually stored in pairs of key and value (the key is what you give to at:—it specifies a slot; the value is what is actually stored at that slot). Notice how we were able to specify not only integers but also strings as both the key and the value. In fact, we can use any kind of object we want as either—the dictionary doesn't care.

Now we can map each key to a value:

```
(x at: 1) printNl !
(x at: 'Two') printNl !
```

which prints respectively:

```
'One'
2
```

We can also ask a dictionary to print itself:

```
x printNl !
```

which prints:

```
Dictionary (1,'One' 2,'Two' 'One',1 'Two',2 )
```

where the first member of each pair is the key, and the second the value.

2.4. Smalltalk dictionary

If you'll remember from the beginning of the chapter, we started out by saying:

```
Smalltalk at: #x put: 0 !
```

This code should look familiar—the at:put: message is how we've been storing information in our own arrays and dictionaries. In a Smalltalk environment the name "Smalltalk" has been preset to point to a dictionary[7] which both you *and* Smalltalk can use. To see how this sharing works, we'll first try to use a variable which Smalltalk doesn't know about:

---

[7] Actually, a SystemDictionary, which is just a Dictionary with some extra hooks to run things when Smalltalk first starts

```
y := 0 !
```

Smalltalk complains because "y" is an unknown variable. Using our knowledge of dictionaries, and taking advantage of our access to Smalltalk's dictionary, we can add it ourselves:

```
Smalltalk at: #y put: 0 !
```

The only mystery left is why we're using "#y" instead of our usual quoted string. This is one of those simple questions whose answer runs surprisingly deep. The quick answer is that "#y" and "'y'" are pretty much the same, except that the former will always be the same object each time you use it, whereas the latter can be a new string each time you do so.[8]

Now that we've added "y" to Smalltalk's dictionary, we try again:

```
y := 1 !
```

It works! Because you've added an entry for "y", Smalltalk is now perfectly happy to let you use this new variable.

If you have some spare time, you can print out the *entire* Smalltalk dictionary with:

```
Smalltalk printNl !
```

As you might suspect, this will print out quite a large list of names! If you get tired of watching Smalltalk grind it out, use your interrupt key (control-C, usually) to bring Smalltalk back to interactive mode.

2.5. Closing thoughts

You've seen how Smalltalk provides you with some very powerful data structures. You've also seen how Smalltalk itself uses these same facilities to implement the language. But this is only the tip of the iceberg—Smalltalk is much more than a collection of "neat" facilities to use.

The objects and methods which are automatically available are only the beginning of the foundation on which you build your programs—Smalltalk allows you to add your own objects and methods into the system, and then use them along with everything else. The art of programming in Smalltalk is the art of looking at your problems in terms of objects, using the existing object types to good effect, and enhancing Smalltalk with new types of objects. Now that you've been exposed to the basics of Smalltalk manipulation, we can begin to look at this object-oriented technique of programming.

---

[8] For more detail, please feel free to skip out to chapter 12 and read the section "Two Flavors of Equality" and the following section "Checking for the Two Types of Equality."

3. The Smalltalk class hierarchy

When programming in Smalltalk, you sometimes need to create new kinds of objects, and define what various messages will do to these objects. In the next chapter we will create some new classes, but first we need to understand how Smalltalk organizes the types and objects it contains. Because this is a pure "concept" chapter, without any actual Smalltalk code to run, we will keep it short and to the point.

3.1. Class Object

Smalltalk organizes all of its classes as a tree hierarchy. At the very top of this hierarchy is class "Object". Following somewhere below it are more specific classes, such as the ones we've worked with—strings, integers, arrays, and so forth. They are grouped together based on their similarities—for instance, types of objects which may be compared as greater or less than each other fall under a class known as "Magnitude".

One of the first tasks when creating a new object is to figure out where within this hierarchy your object falls. Coming up with an answer to this problem is at least as much art as science, and there are no hard-and-fast rules to nail it down. We'll take a look at three kinds of objects to give you a feel for how this organization matters.

3.2. Animals

Imagine that we have three kinds of objects, representing "Animals", "Parrots", and "Pigs". Our messages will be "eat", "sing", and "snort". Our first pass at inserting these objects into the Smalltalk hierarchy would organize them like:

```
Object
    Animals
    Parrots
    Pigs
```

This means that Animals, Parrots, and Pigs are all direct descendants of "Object", and are not descendants of each other.

Now we must define how each animal responds to each kind of message.

```
Animals
    eat--Say "I have now eaten"
    sing--Error
    snort--Error
Parrots
    eat--Say "I have now eaten"
    sing--Say "Tweet"
    snort--Error
Pigs
    eat--Say "I have now eaten"
    sing--Error
    snort--Say "Oink"
```

Notice how we kept having to indicate an action for "eat". An experienced object designer would immediately recognize this as a clue that we haven't set up our hierarchy correctly. Let's try a different organization:

```
Animals
    Parrots
        Pigs
```

That is, Parrots inherit from Animals, and Pigs from Parrots. Now Parrots inherit all of the actions from Animals, and Pigs from both Parrots *and* Animals. Because of this inheritance, we may now define a new set of actions which spares us the redundancy of the previous set:

```
Animals
    eat--Say "I have now eaten"
    sing--Error
    snort--Error
Parrots
    sing--Say "Tweet"
Pigs
    snort--Say "Oink"
```

Because Parrots and Pigs both inherit from Animals, we have only had to define the "eat" action once. However, we have made one mistake in our class setup—what happens when we tell a Pig to "sing"? It says "Tweet", because we have put Pigs as an inheritor of Parrots. Let's try one final organization:

```
Animals
    Parrots
    Pigs
```

Now Parrots and Pigs inherit from Animals, but not from each other. Let's also define one final pithy set of actions:

```
Animals
    eat--Say "I have eaten"
Parrots
    sing--Say "Tweet"
Pigs
    snort--Say "Oink"
```

The change is just to leave out messages which are inappropriate. If Smalltalk detects that a message is not known by an object or any of its ancestors, it will automatically give an error—so you don't have to do this sort of thing yourself. Notice that now sending "sing" to a Pig does indeed *not* say "Tweet"—it will cause a Smalltalk error instead.

3.3. The bottom line of the class hierarchy

The goal of the class hierarchy is to allow you to organize objects into a relationship which allows a particular object to *inherit* the code of its ancestors. Once you have identified an effective organization of types, you should find that a particular technique need only be implemented once, then inherited by the children below. This keeps your code smaller, and allows you to fix a bug in a particular algorithm in only once place—then have all users of it just inherit the fix.

You will find your decisions for adding objects change as you gain experience. As you become more familiar with the existing set of objects and messages, your selections will increasingly "fit in" with the existing ones. But even a Smalltalk "pro" stops and thinks carefully at this stage—so don't be daunted if your first choices seem difficult and error-prone.

4. Creating a new class of objects

With the basic techniques presented in the preceding chapters, we're ready do our first real Smalltalk program. In this chapter we will construct three new types of objects (known as "classes"), using the Smalltalk technique of *inheritance* to tie the classes together, create new objects belonging to these classes (known as creating *instances* of the class), and send messages to these objects.

We'll exercise all this by implementing a toy home-finance accounting system. We will keep track of our overall cash, and will have special handling for our checking and savings accounts. From this point on, we will be defining classes which will be used in future chapters. Since you will probably not be running this whole tutorial in one Smalltalk session, it would be nice to save off the state of Smalltalk and resume it without having to retype all the previous examples. To save the current state of GNU Smalltalk, type:

```
Smalltalk snapshot: 'myimage.img' !
```

and from your shell, to later restart Smalltalk from this "snapshot":

```
% mst -I myimage.img
```

Such a snapshot currently takes a little over 300K bytes, and contains all variables, classes, and definitions you have added.

4.1. Creating a new class

Guess how you create a new class? This should be getting monotonous by now—by sending a message to an object. The way we create our first "custom" class is by sending the following message:

```
Object subclass: #Account
    instanceVariableNames: 'balance'
    classVariableNames: ''
    poolDictionaries: ''
    category: nil !
```

Quite a mouthful, isn't it? Most people end up customizing their editor to pop this up at a push of a button. But conceptually, it isn't really that bad. The Smalltalk variable "Object" is bound to the grand-daddy of all classes on the system. What we're doing here is telling the "Object" class that we want to add to it a subclass known as "Account". The other parts of the message can be ignored, but "instanceVariableNames: 'balance'" tells it that each object in this subclass will have a hidden variable named "balance".[9]

4.2. Documenting the class

The next step is to associate a description with the class. You do this by sending a message to the new class:

```
Account comment: 'I represent a place to deposit and withdraw money' !
```

A description is associated with *every* Smalltalk class, and it's considered good form to add a description to each new class you define. To get the description for a given class:

```
(Account comment) printNl !
```

And your string is printed back to you. Try this with class Integer, too:

```
(Integer comment) printNl !
```

4.3. Defining a method for the class

We have created a class, but it isn't ready to do any work for us—we have to define some messages which the class can process first. We'll start at the beginning by defining methods for instance creation:

---

[9] In case you're having a hard time making out the font, the "''"'s after classVariableNames: and poolDictionaries: are a pair of single quotes—an empty string.

```
!Account class methodsFor: 'instance creation'!

new
        | r |

        r := super new.
        r init.
        ^r
!!
```

Again, programming your editor to do this is recommended. The important points about this are "Account class", which means that we are defining messages which are to be sent to the Account class itself. "methodsFor: 'instance creation'" is more documentation support; it says that all of the methods defined will be to support creating objects of type Account. Finally, the text starting with "new" and ending with "!!" defined what action to take for the message "new". When you enter this definition, GNU Smalltalk will simply give you another prompt. You method has been compiled in and is ready for use. GNU Smalltalk is pretty quiet on successful method definitions--but you'll get plenty of error messages if there's a problem!

This is also the first example where we've had to use more than one statement, and thus a good place to present the statement separator—the ".". Like Pascal, and unlike C, statements are *separated* rather than terminated. Thus you need only use the "." when you have finished one statement and are starting another. This is why our last statement, "^r", does not have a "." following.

The best way to describe how this method works is to step through it. Imagine we sent a message to the new class Account with the command line:

```
Account new !
```

"Account" receives the message "new" and looks up how to process this message. It finds our new definition, and starts running it. The first line, "| r |", creates a *local variable* named "r" which can be used as a placeholder for the objects we create. "r" will go away as soon as the message is done being processed.

The first real step is to actually create the object. The line "r := super new" does this using a fancy trick. The word "super" stands for the same object that the message "new" was originally sent to (remember—it's "Account"), except that when Smalltalk goes to search for the methods, he starts one level *higher* up in the hierarchy than the current level. So for a method in the Account class, this is the Object class (because the class Account inherits from is Object—go back and look at how we created the Account class), and the Object class' methods then execute some code in response to the "new" message. As it turns out, Object will do the actual creation of the object when sent a "new" message.

One more time in slow motion: the Account method "new" wants to do some fiddling about when new objects are created, but he also wants to let his parent do some work with a method of the *same name.* By saying "r := super new" he is letting his parent create the object, and then he is attaching it to the variable "r". So after this line of code executes, we have a brand new object of type Account, and "r" is bound to it. You will understand this better as time goes on, but for now scratch your head once, accept it as a recipe, and keep going.

We have the new object, but we haven't set it up correctly. Remember the hidden variable "balance" which we saw in the beginning of this chapter? "super new" gives us the object with the "balance" field containing nothing—we want our balance field to start at 0.[10] So what we need to do is ask the object to set itself up. By saying "r init", we are sending the "init" message to our new Account. We'll define this method in the next section—for now just assume that sending the "init" message will get our Account set up.

Finally, we say "^r". In English, this is "return what r is attached to". This means that whoever sent "Account" the "new" message will get back this brand new account. At the same time, our temporary

[10] And unlike C, Smalltalk draws a distinction between 0 and nil. nil is the "nothing" object, and you will receive an error if you try to do, say, math on it. It really does matter that we initialize our instance variable to the number 0 if we wish to do math on it in the future.

variable "r" ceases to exist.

4.4. Defining an instance method

We need to define the "init" method for our Account objects, so that our "new" method defined above will work. Here's the Smalltalk code:

```
!Account methodsFor: 'instance initialization'!
init
        balance := 0
!!
```

It looks quite a bit like the previous method definition, except that the first one said "Account class methodsFor:...", and ours says "Account methodsFor:...". The difference is that the first one defined a method for messages sent directly to "Account", but the second one is for messages which are sent to Account objects *once they are created.*

The method named "init" has only one line, "balance := 0". This initializes the hidden variable "balance" (actually called an *instance variable*) to zero, which makes sense for an account balance. Notice that the method doesn't end with "^r" or anything like it—this method doesn't return a value to the message sender. When you do not specify a return value, Smalltalk defaults the return value to the object currently executing. For clarity of programming, you might consider explicitly returning "self" in cases where you intend the return value to be used.[11]

4.5. Looking at our Account

Let's create an *instance* of class Account:

```
Smalltalk at: #a put: (Account new) !
```

Can you guess what this does? The "Smalltalk at: #a put: <something>" hearkens back to chapter 2—it creates a Smalltalk variable. And the "Account new" creates a new Account, and returns it. So this line creates a Smalltalk variable named "a", and attaches it to a new Account—all in one line.

Let's take a look at the Account object we just created:

```
a printNl !
```

It prints:

```
an Account
```

Hmmm... not very informative. The problem is that we didn't tell our Account how to print itself, so we're just getting the default system "printNl" method—which tells what the object *is,* but not what it *contains.* So clearly we must add such a method:

```
!Account methodsFor: 'printing'!
printOn: stream
        super printOn: stream.
        ' with balance: ' printOn: stream.
        balance printOn: stream
!!
```

Now give it a try again:

```
a printNl !
```

which prints:

```
an Account with balance: 0
```

This may seem a little strange. We added a new method, printOn:, and our printNl message starts behaving

[11] And why didn't the designers default the return value to nil? Perhaps they didn't appreciate the value of void functions. After all, at the time Smalltalk was being designed, C didn't even have a void data type.

differently. It turns out that the printOn: message is the central printing function—once you've defined it, all of the other printing methods end up calling it. Its argument is a place to print to—quite often it is the variable "stdout". This variable is usually hooked to your terminal, and thus you get the printout to your screen.

The "super printOn: stream" lets our parent do what it did before—print out what our type is. The "an Account" part of the printout came from this. "' with balance: ' printOn: stream" creates the string " with balance: ", and prints it out to the stream, too. Finally, "balance printOn: stream" asks whatever object is hooked to the "balance" variable to print itself to the stream. We set "balance" to 0, so the 0 gets printed out.

4.6. Moving money around

We can now create accounts, and look at them. As it stands, though, our balance will always be 0—what a tragedy! Our final methods will let us deposit and spend money. They're very simple:

```
!Account methodsFor: 'moving money'!
spend: amount
        balance := balance - amount
!
deposit: amount
        balance := balance + amount
!!
```

With these methods you can now deposit and spend amounts of money. Try these operations:

```
a deposit: 125!
a deposit: 20!
a printNl!
a spend: 10!
a printNl!
```

4.7. Specialized objects

We now have a generic concept, "Account". We can create them, check their balance, and move money in and out of them. They provide a good foundation, but leave out important information that particular types of accounts might want. In the next chapter, we'll take a look at fixing this problem using *subclasses.*

5. Two Subclasses for the Account Class

This chapter continues from the previous chapter in demonstrating how one creates classes and subclasses in Smalltalk. In this chapter we will create two special subclasses of Account, known as Checking and Savings. We will continue to *inherit* the capabilities of Account, but will tailor the two kinds of objects to better manage particular kinds of accounts.

5.1. The Savings class

We create the Savings class as a subclass of Account. It holds money, just like an Account, but has an additional property that we will model: it is paid interest based on its balance. We create the class Savings as a subclass of Account:

```
Account subclass: #Savings
        instanceVariableNames: 'interest'
        classVariableNames: ''
        poolDictionaries: ''
        category: nil !
```

The instance variable "interest" will accumulate interest paid. Thus, in addition to the "spend:" and "deposit:" messages which we inherit from our parent, Account, we will need to define a method to add in *interest* deposits, and a way to clear the interest variable (which we would do yearly, after we have paid taxes). We first define a method for allocating a new account—we need to make sure that the interest field starts at 0.

```
!Savings methodsFor: 'initialization'!
init
        interest := 0.
        ^ super init
!!
```

Recall that the parent took care of the "new" message, and created a new object of the appropriate size. After creation, the parent also sent an "init" message to the new object. As a subclass of Account, the new object will receive the "init" message first; it sets up its own instance variable, and then passes the "init" message up the chain to let its parent take care of its part of the initialization.

With our new "Savings" account created, we can define two methods for dealing specially with such an account:

```
!Savings methodsFor: 'interest'!
interest: amount
        interest := interest + amount.
        self deposit: amount
!
clearInterest
        | oldinterest |

        oldinterest := interest.
        interest := 0.
        ^oldinterest
!!
```

The first method says that we add the "amount" to our running total of interest. The line "self deposit: amount" tells Smalltalk to send ourselves a message, in this case "deposit: amount". This then causes Smalltalk to look up the method for "deposit:", which it finds in our parent, Account. Executing this method then updates our overall balance.[12]

[12] "self" is much like "super", except that "self" will start looking for a method at the bottom of the type hierarchy for the object, but "super" starts looking one level *up* from the current level. Thus, using "super" *forces* inheritance, but "self" will find the first definition of the message which it can.

One may wonder why we don't just replace this with the simpler "balance := balance + amount". The answer lies in one of the philosophies of object-oriented languages in general, and Smalltalk in particular. Our goal is to encode a technique for doing something *once* only, and then re-using that technique when needed. If we had directly encoded "balance := balance + amount" here, there would have been *two* places that knew how to update the balance from a deposit. This may seem like a useless difference. But consider if later we decided to start counting the number of deposits made. If we had encoded "balance := balance + amount" in each place that needed to update the balance, we would have to hunt each of them down in order to update the count of deposits. By sending "self" the message "deposit:", we need only update this method *once;* each sender of this message would then automatically get the correct up-to-date technique for updating the balance.

The second method, "clearInterest", is simpler. We create a temporary variable "oldinterest" to hold the current amount of interest. We then zero out our interest to start the year afresh. Finally, we return the old interest as our result, so that our year-end accountant can see how much we made.[13]

5.2. The Checking class

Our second subclass of Account represents a checking account. We will keep track of two facets:

- What check number we are on
- How many checks we have left in our checkbook

We will define this as another subclass of Account:

```
Account subclass: #Checking
    instanceVariableNames: 'checknum checksleft'
    classVariableNames: ''
    poolDictionaries: ''
    category: nil !
```

We have two instance variables, but we really only need to initialize one of them—if there are no checks left, the current check number can't matter. Remember, our parent class Account will send us the "init" message. We don't need our own class-specific "new" function, since our parent's will provide everything we need.

```
!Checking methodsFor: 'Initialization'!
init
    checksleft := 0.
    ^super init
!!
```

As in Savings, we inherit most of abilities from our superclass, Account. For initialization, we leave "checknum" alone, but set the number of checks in our checkbook to zero. We finish by letting our parent class do its own initialization.

5.3. Writing checks

We will finish this chapter by adding a method for spending money through our checkbook. The mechanics of taking a message and updating variables should be familiar:

[13] Of course, in a real accounting system we would never discard such information—we'd probably throw it into a Dictionary object, indexed by the year that we're finishing. The ambitious might want to try their hand at implementing such an enhancement.

```
!Checking methodsFor: 'spending'!
newChecks: number count: checkcount
        checknum := number.
        checksleft := checkcount
!

writeCheck: amount
        | num |

        num := checknum.
        checknum := checknum + 1.
        checksleft := checksleft - 1.
        self spend: amount.
        ^ num
!!
```

"newChecks:" fills our checkbook with checks. We record what check number we're starting with, and update the count of the number of checks in the checkbook.

"writeCheck:" merely notes the next check number, then bumps up the check number, and down the check count. The message "self spend: amount" resends the message "spend:" to our own object. This causes its method to be looked up by Smalltalk. The method is then found in our parent class, Account, and our balance is then updated to reflect our spending.

You can try the following examples:

```
Smalltalk at: #c put: (Checking new) !
c printNl !
c deposit: 250 !
c printNl !
c newChecks: 100 count: 50 !
c printNl !
(c writeCheck: 32) printNl !
c printNl !
```

For amusement, you might want to add a printOn: message to the checking class so you can see the checking-specific information.

In this chapter, you have seen how to create *subclasses* of your own classes. You have added new methods, and *inherited* methods from the parent classes. These techniques provide the majority of the structure for building solutions to problems. In the following chapters we will be filling in details on further language mechanisms and types, and providing details on how to debug software written in Smalltalk.

6. Code blocks

The Account/Saving/Checking example from the last chapter has several deficiencies. It has no record of the checks and their values. Worse, it allows you to write a check when there are no more checks—the Integer value for the number of checks will just calmly go negative! To fix these problems we will need to introduce more sophisticated control structures.

6.1. Conditions and decision making

Let's first add some code to keep you from writing too many checks. We will simply update our current method for the Checking class; if you have entered the methods from the previous chapters, the old definition will be overridden by this new one.

```
!Checking methodsFor: 'spending'!
writeCheck: amount
        | num |

        (checksleft < 1)
                ifTrue: [ ^self error: 'Out of checks' ].
        num := checknum.
        checknum := checknum + 1.
        checksleft := checksleft - 1.
        self spend: amount
        ^ num
!!
```

The two new lines are:

```
(checksleft < 1)
        ifTrue: [ ^self error: 'Out of checks' ].
```

At first glance, this appears to be a completely new structure. Look again! The only new construct is the square brackets.

The first line is a simple *boolean* expression. "checksleft" is our integer, as initialized by our Checking class. It is sent the message "<", and the argument 1. The current number bound to "checksleft" compares itself against 1, and returns a boolean object telling whether it is less than 1.

Now this boolean—being either true or false—is sent the message "IfTrue:", with an argument which is called a *code block.* A code block is an object, just like any other. But instead of holding a number, or a Set, it holds executable statements.

So what does a boolean *do* with a code block which is an argument to a ifTrue: message? It depends on which boolean! If the object is the "true" object, it executes the code block it has been handed. If it is the "false" object, it returns without executing the code block. So the traditional "conditional construct" has been replaced in Smalltalk with boolean objects which execute the indicated code block or not, depending on their truth-value.[14]

In the case of our example, the actual code within the block sends an error message to the current object. error: is handled by the parent class Object, and will pop up an appropriate complaint when the user tries to write too many checks. In general, the way you handle a fatal error in Smalltalk is to send an error message to yourself (through the "self" pseudo-variable), and let the error handling mechanisms inherited from the Object class take over.

As you might guess, there is also an ifFalse: message which booleans accept. It works exactly like ifTrue:, except that the logic has been reversed; a boolean "false" will execute the codeblock, and a boolean "true" will not.

You should take a little time to play with this method of representing conditionals. You can run your checkbook, but can also invoke the conditional functions directly:

[14] It is interesting to note that because of the way conditionals are done, conditional constructs are not part of the Smalltalk language—they are merely a defined behavior for the Boolean class of objects.

```
true ifTrue: [ 'Hello, world!' printNl ] !
false ifTrue: [ 'Hello, world!' printNl ] !
true ifFalse: [ 'Hello, world!' printNl ] !
false ifFalse: [ 'Hello, world!' printNl ] !
```

6.2. Iteration and collections

Now that we have some sanity checking in place, it remains for us to keep a log of the checks we write. We will do so by adding a Dictionary object to our Checking class, logging checks into it, and providing some messages for querying our check-writing history. But this enhancement brings up a very interesting question—when we change the "shape" of an object (in this case, by adding a new *instance variable* to the Checking class—our dictionary), what happens to the existing class, and its objects?

The answer is that the old objects continue to exist with their current shape.[15] New objects will have the new shape. As this can lead to *very* puzzling behavior, it is usually best to eradicate all of the old objects, and then implement your changes. If this were more than a toy object accounting system, this would probably entail saving the objects off, converting to the new class, and reading the objects back into the new format. For now, we'll just ignore what's currently there, and define our latest Checking class.

```
Account subclass: #Checking
    instanceVariableNames: 'checknum checksleft history'
    classVariableNames: ''
    poolDictionaries: ''
    category: nil !
```

This is the same syntax as the last time we defined a checking account, except that we have *three* instance variables—the "checknum" and "checksleft" which have always been there, and our new "history" variable. We must now feed in our definitions for each of the messages our object can handle, since we are basically defining a new class under an old name.[16] Go ahead and do this now—the methods are in chapter 5. We are using the same Account class, so you only need to do the Checking methods.

With our new Checking instance variable, we are all set to start recording our checking history. Our first change will be in the "init" message handling:

```
!Checking methodsFor: 'initialization'!
init
    checksleft := 0.
    history := Dictionary new.
    ^ super init
!!
```

This provides us with a Dictionary, and hooks it to our new "history" variable.

Our next method records each check as it's written. The method is a little more involved, as we've added some more sanity checks to the writing of checks.

[15] This is the case in GNU Smalltalk. Other implementations will refuse to redefine the class until *all* of its instances have been hunted down and eradicated!

[16] Technically, GNU Smalltalk has associated us with the existing *class* methods, but with a new set of *instance* methods. It is often simpler to work as if you had to define everything, rather than trying to take advantage of this.

```
!Checking methodsFor: 'spending'!
writeCheck: amount
    | num |

    "Sanity check that we have checks left in our checkbook"
    (checksleft < 1)
        ifTrue: [ ^self error: 'Out of checks' ].

    "Make sure we've never used this check number before"
    num := checknum.
    (history includesKey: num)
        ifTrue: [ ^self error: 'Duplicate check number' ].

    "Record the check number and amount"
    history at: num put: amount.

    "Update our next checknumber, checks left, and balance"
    checknum := checknum + 1.
    checksleft := checksleft - 1.
    self spend: amount.
    ^ num
!!
```

We have added three things to our latest version of writeCheck:. First, since our routine has become somewhat involved, we have added comments. In Smalltalk, single quotes are used for strings; double quotes enclose comments. We have added comments before each section of code.

Second, we have added a sanity check on the check number we propose to use. Dictionary objects respond to the includesKey: message with a boolean, depending on whether something is currently stored under the given key in the dictionary. If the check number is already used, the error: message is sent to our object, aborting the operation.

Finally, we add a new entry to the dictionary. We have already seen the at:put: message from chapter 2. Our use here simply associates a check number with an amount of money spent.[17] With this, we now have a working Checking class, with reasonable sanity checks and per-check information.

Let us finish the chapter by enhancing our ability to get access to all this information. We will start with some simple print-out functions.

---

[17] You might start to wonder what one would do if you wished to associate *two* pieces of information under one key. Say, the value and who the check was written to. There are several ways; the best would probably be to create a new, custom object which contained this information, and then store this object under the check number key in the dictionary. It would also be valid (though probably over-kill) to store a dictionary as the value—and then store as many pieces of information as you'd like under each slot!

```
!Checking methodsFor: 'printing'!
printOn: stream
        super printOn: stream.
        ', checks left: ' printOn: stream.
        checksleft printOn: stream.
        ', checks written: ' printOn: stream.
        (history size) printOn: stream.
!
check: num
        | c |
        c := history at: num ifAbsent: [ ^self error: 'No such check #' ].
        ^c
!!
```

There should be very few surprises here. We format and print our information, while letting our parent classes handle their own share of the work. When looking up a check number, we once again take advantage of the fact that blocks of executable statements are an object; in this case, we are using the at:ifAbsent: message supported by the Dictionary class. If the requested key value is not found in the dictionary, the code block is executed. This allows us to customize our error handling, as the generic error would only tell the user "key not found".

While we can look up a check if we know its number, we have not yet written a way to "riffle through" our collection of checks. The following function loops over the checks, printing them out one per line. Because there is currently only a single numeric value under each key, this might seem wasteful. But we have already considered storing multiple values under each check number, so it is best to leave some room for each item. And, of course, because we are simply sending a printing message to an object, we will not have to come back and re-write this code so long as the object in the dictionary honors our printNl/printOn: messages.

```
!Checking methodsFor: 'printing'!
printChecks
        history associationsDo: [:assoc|
                (assoc key) print.
                ' - ' print.
                (assoc value) printNl.
        ]
!!
```

We still see a code block object being passed to the dictionary, but ":assoc|" is something new. A code block can optionally receive *arguments.* In this case, the argument is the key/value pair, known in Smalltalk as an *Association.* This is the way that a dictionary object stores its key/value pairs internally. In fact, when you sent an at:put: message to a dictionary object, the first thing it does is pack them into a new object from the Association class. If you only wanted the value portion, you could call history with a do: message instead.

Our code merely uses the "key" and "value" messages to ask the association for the two values. We then invoke our printing interface upon them. While the printNl message implicitly uses "stdout", we don't want a newline until the end, so the "print" message is used instead. It is pretty much the same as "printNl", except it doesn't add a newline.

It is important that you be clear on the relationship between an Association and the argument to a code block. In this example, we passed a associationsDo: message to a dictionary. A dictionary invokes the passed code block with an Association when processing an associationsDo: message. But code blocks can receive *any* type of argument—the type is determined by the code which invokes the code block; Dictionary, in this case. In the next chapter we'll see more on how code blocks are used; we'll also look at how you can invoke code blocks in your own code.

7. Code blocks, part two

In the last chapter, we looked at how code blocks could be used to build conditional expressions, and how you could iterate across all entries in a collection.[18] We built our own code blocks, and handed them off for use by system objects. But there is nothing magic about invoking code blocks; your own code will often need to do so. This chapter will shows some examples of loop construction in Smalltalk, and then demonstrate how you invoke code blocks for yourself.

7.1. Integer loops

Integer loops are constructed by telling a number to drive the loop. Try this example to count from 1 to 20:

```
1 to: 20 do: [:x| x printNl ] !
```

There's also a way to count up by more than one:

```
1 to: 20 by: 2 do: [:x| x printNl ] !
```

Finally, counting down is done with a negative interval:

```
20 to: 1 by: -1 do: [:x| x printNl ] !
```

7.2. Intervals

It is also possible to represent a range of numbers as a standalone object. This allows you to represent a range of numbers as a single object, which can be passed around the system.

```
Smalltalk at: #i put: (Interval from: 5 to: 10) !
i printNl !
i do: [:x| x printNl] !
```

As with the integer loops, the Interval class can also represent steps greater than 1. It is done much like it was for our numeric loop above:

```
i := (Interval from: 5 to: 10 by: 2)
i printNl !
i do: [:x| x printNl] !
```

7.3. Invoking code blocks

Let us revisit the checking example and add a method for scanning only checks over a certain amount. This would allow our user to find "big" checks, by passing in a value below which we will not invoke their function. We will invoke their code block with the check number as an argument; they can use our existing check: message to get the amount.

```
!Checking methodsFor: 'scanning'!
checksOver: amount do: aBlock
        history associationsDo: [:assoc|
                ((assoc value) > amount)
                        ifTrue: [aBlock value: (assoc key)]
        ]
!!
```

The structure of this loop is much like our printChecks message from chapter 6. However, in this case we consider each entry, and only invoke the supplied block if the check's value is greater than the specified amount. The line:

---

[18] The do: message is understood by most types of Smalltalk collections. It works for the Dictionary class, as well as sets, arrays, strings, intervals, linked lists, bags, and streams. The associationsDo: message works only with dictionaries. The difference is that do: passes only the *value* portion, while associationsDo: passes the entire *key/value* pair in an Association object.

```
ifTrue: [aBlock value: (assoc key)]
```

invokes the user-supplied block, passing as an argument the association's key, which is the check number. The value: message, when received by a code block, causes the code block to execute. Code blocks take "value", "value:", "value:value:", and "value:value:value:" messages, so you can pass from 0 to 3 arguments to a code block.[19] You might find it puzzling that an association takes a "value" message, and so does a code block. Remember, each object can do its own thing with a message. A code block gets *run* when it receives a "value" message. An association merely returns the value part of its key/value pair. The fact that both take the same message is, in this case, coincidence.

Let's quickly set up a new checking account with $250 (wouldn't this be nice in real life?) and write a couple checks. Then we'll see if our new method does the job correctly:

```
Smalltalk at: #mycheck put: (Checking new) !
mycheck deposit: 250 !
mycheck newChecks: 100 count: 40 !
mycheck writeCheck: 10 !
mycheck writeCheck: 52 !
mycheck writeCheck: 15 !
mycheck checksOver: 1 do: [:x| printNl] !
mycheck checksOver: 17 do: [:x| printNl] !
mycheck checksOver: 200 do: [:x| printNl] !
```

We will finish this chapter with an alternative way of writing our checksOver: code. In this example, we will use the message select: to pick the checks which exceed our value, instead of doing the comparison ourselves. We can then invoke the new resulting collection against the user's code block. Unlike our previous definition of checksOver:do:, this one passes the user's code block the association, not just a check number. How could this code be rewritten to remedy this, while still using select:?

```
!Checking methodsFor: 'scanning'!
checksOver: amount do: aBlock
        | chosen |
        chosen := history select: [:amt| amt > amount].
        chosen associationsDo: aBlock
!!
```

You can use the same set of tests that we ran above. Notice that our code block:

```
[:x| x printNl]
```

now prints out an Association. This has the very nice effect--with our old method, we were told which check numbers were above a given amount. With this new method, we get the check number and amount in the form of an Association. When we print an association, since the key is the check number and the value is the check amount, we get a list of checks over the amount in the format:

```
CheckNum -> CheckVal
```

[19] There is also a valueWithArguments: message which accepts an array holding as many arguments as you would like.

8. When Things Go Bad

So far we've been working with examples which work the first time. If you didn't type them in correctly, you probably received a flood of unintelligible complaints. You probably ignored the complaints, and typed the example again.

When developing your own Smalltalk code, however, these messages are the way you find out what went wrong. Because your objects, their methods, the error printout, and your interactive environment are all contained within the same Smalltalk session, you can use these error messages to debug your code using very powerful techniques.

8.1. A Simple Error

First, let's take a look at a typical error. Type:

```
7 plus: 1 !
```

This will print out:

```
7 did not understand selector 'plus:'

UndefinedObject>>#executeStatements
UndefinedObject>>nil
```

The first line is pretty simple; we sent a message to the "7" object which was not understood; not surprising since the "plus" operation should have been "+". The two remaining lines reflect the way the GNU Smalltalk invokes code which we type to our command prompt; it generates a block of code which is invoked via an internal function "executeStatements". Thus, this output tells you that you directly typed a line which sent an invalid message to the "7" object.

The last two lines of the error output are actually a stack backtrace. The most recent call is the one nearer the top of the screen. In the next example, we will cause an error which happens deeper within an object.

8.2. Nested Calls

Type the following lines:

```
Smalltalk at: #x put: (Dictionary new) !
x at: 1 !
```

The error you receive will look like:

```
Dictionary new: 32 "<0x33788>" error: key not found

MethodContext>>#value
Dictionary>>#at:ifAbsent:
Dictionary>>#at
UndefinedObject>>#executeStatements
UndefinedObject>>nil
```

The error itself is pretty clear; we asked for something within the Dictionary which wasn't there. The object which had the error is identified as "Dictionary new: 32". A Dictionary's default size is 32; thus, this is the object we created with "Dictionary new".

The stack backtrace shows us the inner structure of how a Dictionary responds to the at: message. Our hand-entered command causes the usual two entries for "UndefinedObject". Then we see a Dictionary object responding to an "at:" message (the "Dictionary>>#at" line). This code called the object with an "at:ifAbsent:" message. All of a sudden, a different object receives a "value" message, and then the error happened.

This isn't quite true; the error happened in the Dictionary object. The mystery is where this "MethodContext" came from. Fortunately, it isn't much of a mystery. The answer lies in what we covered in the last two chapters: code blocks.

A very common way to handle errors in Smalltalk is to hand down a block of code which will be called when an error occurs. For the Dictionary code, the "at:" message passes in a block of code to the at:ifAbsent: code to be called when "at:ifAbsent:" can't find the given key. Thus, without even looking at the code for Dictionary itself, we can guess that the Dictionary "at:" message handling looks something like:

```
at: key ifAbsent: errCodeBlock
    ...look for key...
    (keyNotFound) ifTrue: [ ^(errCodeBlock value) ]
    ...

at: key
    ^self at: key ifAbsent: [^self error: 'key not found']
```

The key is that we see in the stack backtrace that at:ifAbsent: is called from at:, and a MethodContext is called to give the error. Once we realize that a MethodContext is just a fancy name for a code block, we can guess that at: handed in a code block to print an error, and at:ifAbsent: used the standard "value" message to invoke this code block when it realized that the requested key couldn't be found in the Dictionary.

It would be nice if each entry on the stack backtrace included source line numbers. Unfortunately, at this point GNU Smalltalk doesn't provide this feature. Of course, you have the source code available....

8.3. Some Shortcomings in GNU Smalltalk

Unfortunately, there are some errors which GNU Smalltalk is not very helpful in detecting. This information applies to the latest version currently available—GNU Smalltalk 1.1.1. Try indexing something which isn't any sort of Collection:

```
(7 at: 99) printNl !
```

One would expect to receive an error,[20] but instead GNU Smalltalk simply returns the receiving object—7. Similarly, one would expect Array bounds to be checked:

```
Smalltalk at: #x put: (Array new: 10) !
(x at: 7 put: 123) printNl !
(x at: 11 put: 1234) printNl !
(x at: 7) printNl !
(x at: 11) printNl !
```

But this example returns no error in GNU Smalltalk.[21] When an assignment to an array slot is correct, the returned value is the value assigned—123 in this case. When you assign outside the array bounds, you will receive the array object itself as the result! Thus, if you accidentally index your array incorrectly, you will have to figure out what happened from a place further in your code where an error crops up because your code was trying to operate upon an *element* of the array, but instead is working on the array itself.

8.4. Looking at Objects

When you are chasing an error, it is often helpful to examine the instance variables of your objects. While strategic "printNl"s will no doubt help, you can look at an object without having to write all the code yourself. The "inspect" message works on any object, and dumps out the values of each instance variable within the object. Thus:

```
Smalltalk at: #x put: (Interval from: 1 to: 5) !
x inspect !
```

displays:

---

[20] And in fact, you *will* receive an error in most Smalltalk systems.

[21] Again, it *will* error in most other Smalltalk implementations.

```
An instance of Interval
  start: 1
  stop: 5
  step: 1
```

There's one thing the object inspector doesn't display—the contents of an *indexed class.* Since we haven't looked at this kind of object yet, we'll leave this to its own chapter.

We'll finish this chapter by emphasizing a technique which has already been covered—the use of the "error:" message in your own objects. As you saw in the case of Dictionary, an object can send itself an error: message with a descriptive string to abort execution and dump a stack backtrace. You should plan on using this technique in your own objects. It can be used both for explicit user-caused errors, as well as in internal sanity checks.

9. Coexisting in the Class Hierarchy

The early chapters of this paper discussed classes in one of two ways. The "toy" classes we developed were rooted at Object; the system-provided classes were treated as immutable entities. While one shouldn't modify the behavior of the standard classes lightly, "plugging in" your own classes in the right place among their system-provided brethren can provide you powerful new classes with very little effort.

This chapter will create two complete classes which enhance the existing Smalltalk hierarchy. The discussion will start with the issue of where to connect our new classes, and then continue onto implementation. Like most programming efforts, the result will leave many possibilities for improvements. The framework, however, should begin to give you an intuition of how to develop your own Smalltalk classes.

9.1. The Existing Class Hierarchy

To discuss where a new class might go, it is helpful to have a map of the current classes. The following is the class hierarchy of GNU Smalltalk 1.1.1. Indentation means that the line inherits from the earlier line with one less level of indentation.[22]

[22] This listing is courtesy of the printHierarchy method supplied by GNU Smalltalk author Steve Byrne. If you have the GNU Smalltalk source, it's in the samples/ directory.

- Object
    - Autoload
    - Behavior
        - ClassDescription
            - Class
            - Metaclass
    - BlockContext
    - Boolean
        - False
        - True
    - CFunctionDescriptor
    - CObject
    - Collection
        - Bag
        - MappedCollection
        - SequenceableCollection
            - ArrayedCollection
                - Array
                - ByteArray
                - CompiledMethod
                - String
                    - Symbol
            - Interval
            - LinkedList
                - Semaphore
            - OrderedCollection
                - SortedCollection
        - Set
            - Dictionary
                - IdentityDictionary
                - SystemDictionary
    - Delay
    - FileSegment
    - Link
        - Process
        - SymLink
    - Magnitude
        - Character
        - Date
        - LookupKey
            - Association
        - Number
            - Float
            - Integer
        - Time
    - Memory
        - ByteMemory
        - WordMemory
    - Message
    - MethodContext
    - MethodInfo
    - ProcessorScheduler
    - SharedQueue
    - Stream

```
      PositionableStream
         ReadStream
         WriteStream
            ReadWriteStream
               FileStream
      Random
      TokenStream
   UndefinedObject
```

While initially a daunting list, you should take the time to hunt down the classes we've examined in this paper so far. Notice, for instance, how an Array is a subclass below the "SequenceableCollection" class. This makes sense; you can walk an Array from one end to the other. By contrast, notice how a Set is at the same level as SequenceableCollection. It doesn't make sense to walk a Set from one end to the other.

A little puzzling is the relationship of a Dictionary to a Set; why is a Dictionary a subclass of a Set? The answer lies in the basic structure of both a Set and a Dictionary. Both hold an unordered collection of objects. For a set, they're *any* objects; for a Dictionary, they are Associations. Thus, Dictionary inherits some of the more basic mechanisms for creating itself, and then adds an extra layer of interpretation.

Finally, look at the treatment of numbers—starting with the class Magnitude. While numbers can be ordered by "less than", "greater than", and so forth, so can a number of *other* objects. Each subclass of Magnitude is such an object. So we can compare characters with other characters, dates with other dates, and times with other times, as well as numbers with numbers.[23]

9.2. Those Darn Arrays

Imagine that you're chasing an array problem, and the lack of a clear bounds check is making it too hard. You could modify the Smalltalk implementation, but perhaps it's in somebody else's directory, so it wouldn't be practical. Why not add a subclass, put some sanity checks in the array indexing, and use our superclass to do all the work?

[23] Ignore LookupKey; its presence appears to be historical.

```
Array variableSubclass: #CheckedArray
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: nil !

!CheckedArray methodsFor: 'bounds checking'!
boundsCheck: index
        ((index < 1) | (index > (self basicSize))) ifTrue: [
                ^self illegalIndex
        ]
!
illegalIndex
        ^self error: 'Illegal index'
!!

!CheckedArray methodsFor: 'basic'!
at: index
        self boundsCheck: index.
        ^super at: index

!
at: index put: val
        self boundsCheck: index.
        ^super at: index put: val
!!
```

Much of the machinery of adding a class should be familiar. Instead of our usual subclass: message, we use a variableSubclass: message. This reflects the underlying structure of an Array object; we'll delay discussing this until the chapter on the nuts and bolts of arrays. In any case, we *inherit* all of the actual knowledge of how to create arrays, reference them, and so forth. All that we do is intercept at: and at:put: messages, and call our common function to validate the array index. The way that we coded the bounds check bears a little examination.

Making a first cut at coding the bounds check, you might have coded the bounds check in CheckedArray's methods twice—once for at:, and again for at:put:. As always, it's preferable to code things *once,* and then re-use them. So we instead add a method for bounds checking "boundsCheck:", and use it for both cases. If we ever wanted to enhance the bounds checking (perhaps enhance the error message to print the offending index value?), we only have to change it in one place.

The actual math for calculating whether the bounds have been violated is a little interesting. The first part of the expression:

```
((index < 1) | (index > (self basicSize)))
```

is true if the index is less than 1, otherwise it's false. This part of the expression thus becomes the boolean object true or false. The boolean object then receives the message "|", and the argument "(index > (self basicSize))". "|" means "or"—we want to OR together the two possible illegal range checks. What is the second part of the expression?[24]

"index" is our argument, an integer; it receives the message ">", and thus will compare itself to the value "self basicSize" returns. While we haven't covered the underlying structures Smalltalk uses to build arrays, we can briefly say that the "basicSize" message returns the number of elements the Array object can

---

[24] Smalltalk also offers an "or:" message, which is different in a subtle way from "|". or: takes a code block, and only invokes the code block if it's necessary to determine the value of the expression. This is analogous to the guaranteed C semantic that "&&" evaluates left-to-right only as far as needed. We could have written the expressions as "((index < 1) or: [index > (self basicSize)])". Since we expect both sides of or: to be false most of the time, there isn't much reason to delay evaluation of either side.

contain. So the index is checked to see if it's less than 1 (the lowest legal Array index) or greater than the highest allocated slot in the Array. If it is either (the "|" operator), the expression is true, otherwise false.

From there it's downhill; our boolean object receives the ifTrue: message, and a code block which will send an error message to the object. Why do we have a separate message just to print the error? For purposes of this example, it's not needed. But one could conceive, in general, of a couple of different sanity checks all sharing the same mechanism for actually printing the error message. So we'll write it this way anyway.

9.3. Adding a New Kind of Number

If we were programming an application which did a large amount of complex math, we could probably manage it with a number of two-element arrays. But we'd forever be writing in-line code for the math and comparisons; it would be much easier to just implement an object class to support the complex numeric type. Where in the class hierarchy would it be placed?

You've probably already guessed—but let's step down the hierarchy anyway. *Everything* inherits from Object, so that's a safe starting point. Complex numbers can not be compared with "<" and ">", and yet we strongly suspect that, since they are numbers, we should place them under the Number class. But Number inherits from Magnitude--how do we resolve this conflict? A subclass can place itself under a superclass which allows some operations the subclass doesn't wish to allow. All that you must do is make sure you intercept these messages and return an error. So we will place our new Complex class under Number, and make sure to disallow comparisons.

One can reasonably ask whether the real and imaginary parts of our complex number will be integer or floating point. In the grand Smalltalk tradition, we'll just leave them as objects, and hope that they respond to numeric messages reasonably. If they don't, the user will doubtless receive errors and be able to track back their mistake with little fuss.

We'll define the four basic math operators, as well as the (illegal) relationals. We'll add printOn: so that the printing methods work, and that should give us our Complex class. The class as presented suffers some limitations, which we'll cover later in the chapter.

```
Number subclass: #Complex
        instanceVariableNames: 'realpart imagpart'
        classVariableNames: ''
        poolDictionaries: ''
        category: nil !
!Complex class methodsFor: 'creating'!
new
        ^self error: 'use real:imaginary:'
!
new: ignore
        ^self new
!
real: r imaginary: i
        ^(super new) setReal: r setImag: i
!!

!Complex methodsFor: 'creating--private'!
setReal: r setImag: i
        realpart := r.
        imagpart := i.
        ^self
!!

!Complex methodsFor: 'basic'!
real
        ^realpart
!
imaginary
        ^imagpart
!!

!Complex methodsFor: 'math'!
+ val
        ^Complex real: (realpart + (val real))
                imaginary: (imagpart + (val imaginary))
!
- val
        ^Complex real: (realpart - (val real))
                imaginary: (imagpart - (val imaginary))
!
* val
        ^Complex real: ((realpart * (val real)) - (imagpart * (val imaginary)))
                imaginary: ((realpart * (val imaginary)) +
                        (imagpart * (val real)))
!
/ val
        | d r i |
        d := ((val real) * (val real)) + ((val imaginary) * (val imaginary)).
        r := ((realpart * (val real)) + (imagpart * (val imaginary))) / d.
        i := ((imagpart * (val real)) - (realpart * (val imaginary))) / d.
        ^Complex real: r imaginary: i
!!

!Complex methodsFor: 'comparison'!
```

```
= val
        ^((realpart = (val real)) & (imagpart = (val imaginary)))
!
> val
        ^self shouldNotImplement
!
>= val
        ^self shouldNotImplement
!
< val
        ^self shouldNotImplement
!
<= val
        ^self shouldNotImplement
!!

!Complex methodsFor: 'printing'!
printOn: aStream
        aStream nextPut: $(.
        realpart printOn: aStream.
        aStream nextPut: $,.
        imagpart printOn: aStream.
        aStream nextPut: $)
!!
```

There should be surprisingly little which is actually new in this example. The printing method uses both printOn: as well as nextPut: to do its printing. While we haven't covered it, it's pretty clear that "$(" generates the ASCII character "(" as an object, and nextPut: puts its argument as the next thing on the stream.

The math operations all generate a new object, calculating the real and imaginary parts, and invoking the Complex class to create the new object. Our creation code is a little more compact than earlier examples; instead of using a local variable to name the newly-created object, we just use the return value and send a message directly to the new object. Our initialization code explicitly returns self; what would happen if we left this off?[25]

9.4. Inheritance and Polymorphism

This is a good time to look at what we've done with the two previous examples at a higher level. With the CheckedArray class, we *inherited* almost all of the functionality of arrays, with only a little bit of code added to address our specific needs. While you may have not thought to try it, all the existing methods for an Array continue to work without further effort—you might find it interesting to ponder why the following still works:

```
Smalltalk at: #a put: (CheckedArray new: 10) !
a at: 5 put: 1234 !
a do: [:i| i printNl ] !
```

The strength of inheritance is that you focus on the incremental changes you make; the things you *don't* change will generally continue to work.

In the Complex class, the value of *polymorphism* was exercised. A Complex number responds to exactly the same set of messages as any other number. If you had handed this code to someone, they would know how to do math with Complex numbers without further instruction. Compare this with C, where a complex number package would require the user to first find out if the complex-add function was

[25] Hint: consider what the default return value is when no explicit value is provided. This was covered in chapter 4.

complex_plus(), or perhaps complex_add(), or add_complex(), or....

9.5. Limitations of the Complex Class

One glaring deficiency is present in the Complex class—what happens if you mix normal numbers with Complex numbers? Currently, the Complex class assumes that it will only interact with other Complex numbers. But this is unrealistic—mathematically, a "normal" number is simply one with an imaginary part of 0. Smalltalk was designed to allow numbers to *coerce* themselves into a form which will work with other numbers.

The system is clever and requires very little additional code. Unfortunately, it would have tripled the amount of explanation required. If you're interested in how coercion works in GNU Smalltalk, you should find the Smalltalk library source, and trace back the execution of the retry:coercing: messages. You want to consider the value which the "generality" message returns for each type of number. Finally, you need to examine the coerce: handling in each numeric class.

10. Smalltalk Streams

Our examples have used a mechanism extensively, even though we haven't discussed it yet. The Stream class provides a framework for a number of data structures, including input and output functionality, queues, and endless sources of dynamically-generated data. A Smalltalk stream is quite similar to the UNIX streams you've used from C. A stream provides a sequential view to an underlying resource; as you read or write elements, the stream position advances until you finally reach the end of the underlying medium. Most streams also allow you to *set* the current position, providing random access to the medium.

10.1. The Output Stream

The examples in this book all work because they write their output to the **stdout** stream. Each class implements the printOn: method, and writes its output to the supplied stream. The printNl method all objects use is simply to send the current object a printOn: message whose argument is **stdout.** You can invoke the standard output stream directly:

```
'Hello, world' printOn: stdout !
stdout inspect !
```

10.2. Your Own Stream

Unlike a pipe you might create in C, the underlying storage of a Stream is under your control. Thus, a Stream can provide an anonymous buffer of data, but it can also provide a stream-like interpretation to an existing array of data. Consider this example:

```
Smalltalk at: #a put: (Array new: 10) !
a at: 4 put: 1234 !
a at: 9 put: 5678 !
Smalltalk at: #s put: (ReadWriteStream on: a) !
s inspect !
s position: 1 !
s inspect !
s nextPut: 11; nextPut: 22 !
(a at: 1) printNl !
a do: [:x| x printNl] !
s position: 2 !
s do: [:x| x printNl] !
s position: 5 !
s do: [:x| x printNl] !
s inspect !
```

The key is the on: message; it tells a stream class to create itself in terms of the existing storage. Because of polymorphism, the object specified by on: does not have to be an Array; any object which responds to numeric at: messages can be used. If you happen to have the CheckedArray class still loaded from the previous chapter, you might try streaming over that kind of array instead.

You're wondering if you're stuck with having to know how much data will be queued in a Stream at the time you create the stream. If you use the right class of stream, the answer is no. A ReadStream provides read-only access to an existing collection. You will receive an error if you try to write to it. If you try to read off the end of the stream, you will also get an error.

By contrast, WriteStream and ReadWriteStream (used in our example) will tell the underlying collection to grow (using the "grow" message) when you write off the end of the existing collection. Thus, if you want to write several strings, and don't want to add up their lengths yourself:

```
Smalltalk at: #s put: (ReadWriteStream on: (String new: 0)) !
s inspect !
'Hello, ' printOn: s !
s inspect !
'world' printOn: s !
s inspect !
s position: 1 !
s inspect !
s do: [:c| c printOn: stdout] !
(s contents) printNl !
```

In this case, we have used a String as the collection for the Stream. The printOn: messages add bytes to the initially empty string. Once we've added the data, you can continue to treat the data as a stream. Alternatively, you can ask the stream to return to you the underlying object. After that, you can use the object (a String, in this example) using its own access methods.

There are many amenities available on a stream object. You can ask if there's more to read with "atEnd". You can query the position with "position", and set it with "position:". You can see what will be read next with "peek", and you can read the next element with "next".

In the writing direction, you can write an element with "nextPut:". You don't need to worry about objects doing a printOn: with your stream as a destination; this operation ends up as a sequence of nextPut:'s to your stream. If you have a collection of things to write, you can use "nextPutAll:" with the collection as an argument; each member of the collection will be written onto the stream. If you want to write an object to the stream several times, you can use "next:put:":

```
Smalltalk at: #s put: (ReadWriteStream on: (Array new: 0)) !
s next: 4 put: 'Hi!' !
s position: 1 !
s do: [:x| x printNl] !
```

10.3. Files

Streams can also operate on files. If you wanted to dump the file "/etc/passwd" to your terminal, you could create a stream on the file, and then stream over its contents:

```
Smalltalk at: #f put: (FileStream open: '/etc/passwd' mode: 'r') !
f do: [:c| c printOn: stdout] !
f position: 30 !
1 to: 25 do: [(f next) printOn: stdout] !
f close !
```

and, of course, you can load Smalltalk source code:

```
FileStream fileIn: '/users/myself/src/source.st' !
```

10.4. Dynamic Strings

Streams provide a powerful abstraction for a number of data structures. Concepts like current position, writing the next position, and changing the way you view a data structure when convenient combine to let you write compact, powerful code. The last example is taken from the actual Smalltalk source code—it shows a general method for making an object print itself onto a a string.

```
printString
    | stream |
    stream := WriteStream on: (String new: 0).
    self printOn: stream.
    ^stream contents
!
```

This method, residing in Object, is inherited by every class in Smalltalk. The first line creates a WriteStream which stores on a String whose length is currently 0. It then invokes the current object with printOn:. As the object prints itself to "stream", the String grows to accommodate new characters. When the object is done printing, the method simply returns the underlying string.

As we've written code, the assumption has been that printOn: would go to the terminal. But replacing a stream to a file (/dev/tty) with a stream to a data structure (String new: 0) works just as well. The last line tells the Stream to return its underlying collection—which will be the string which has had all the printing added to it. The result is that the printString message returns an object of the String class whose contents are the printed representation of the object receiving the printString message.

11. How Arrays Work

Smalltalk provides a very adequate selection of predefined classes from which to choose. Eventually, however, you will find the need to code a new basic data structure. Because Smalltalk's most fundamental storage allocation facilities are arrays, it is important that you understand how to use them to gain efficient access to this kind of storage.

11.1. The Array Class

Our examples have already shown the Array class, and its use is fairly obvious. For many applications, it will fill all your needs—when you need an array in a new class, you keep an instance variable, allocate a new Array and assign it to the variable, and then send array accesses via the instance variable.

This technique even works for string-like objects, although it is wasteful of storage. An Array object uses a *Smalltalk pointer* for each slot in the array; its exact size is transparent to the programmer, but you can generally guess that it'll be roughly the word size of your machine.[26] For storing an array of characters, therefore, an Array works but is inefficient.

11.2. Arrays at a Lower Level

So let's step down to a lower level of data structure. a ByteArray is much like an Array, but each slot holds only an integer from 0 to 255—and each slot uses only a byte of storage. If you only needed to store small quantities in each array slot, this would therefore be a much more efficient choice than an Array. As you might guess, this is the type of array which a String uses.

Aha! But when you go back to chapter 9 and look at the Smalltalk hierarchy, you notice that String does *not* inherit from ByteArray. To see why, we must delve down yet another level, and arrive at the basic methods for compiling a class.

For most example classes, we've used the message:

```
subclass:
instanceVariableNames:
classVariableNames:
poolDictionaries:
category:
```

But when we implemented our CheckedArray example, we used "variableSubclass:" instead of just "subclass:". The choice of these two kinds of class creation (and a third we'll show shortly) defines the fundamental structure of Smalltalk objects created within a given class. Let's consider the differences in the next three sub-sections.

11.2.1. subclass:

This kind of class creation specifies the simplest Smalltalk object. The object consists only of the storage needed to hold the instance variables. In C, this would be a simple structure with zero or more scalar fields.[27]

11.2.2. variableSubclass:

This type of class is a superset of a subclass:. Storage is still allocated for any instance variables, but the objects of the class must be created with a new: message. The number passed as an argument to new: causes the new object, in addition to the space for instance variables, to also have that many slots of storage allocated. The analog in C would be to have a structure with some scalar fields, followed at its end by an array of the requested size of pointers.

---

[26] 32 bits for most ports of GNU Smalltalk.

[27] C requires one or more; zero is allowed in Smalltalk.

11.2.3. variableByteSubclass:

This is a special case of variableSubclass:; the storage allocated as specified by new: is an array of *bytes*. The analog in C would be a structure with scalar fields, followed by an array of *char.*

11.3. Accessing These New Arrays

You already know how to access instance variables—by name. But there doesn't seem to be a name for this new storage. The way an object accesses it is to send itself array-type messages—at:, at:put:, and so forth.

The problem is when an object wants to add a new level of interpretation to the at: and at:put: messages. Consider a Dictionary—it is a variableSubclass: type of object, but its at: message is in terms of a *key,* not an integer index of its storage. Since it has redefined the at: message, how does it access its fundamental storage?

The answer is that Smalltalk has defined basicAt: and basicAt:put:, which will access the basic storage even when the at: and at:put: messages have been defined to provide a different abstraction.

11.4. An Example

This can get pretty confusing in the abstract, so let's do an example to show how it's pretty simple in practice. Smalltalk arrays tend to start at 1; let's define an array type whose permissible range is arbitrary.

```
ArrayedCollection variableSubclass: 'RangedArray'
    instanceVariableNames: 'base'
    classVariableNames: ''
    poolDictionaries: ''
    category: nil !
RangedArray comment: 'I am an Array whose base is arbitrary' !
!RangedArray class methodsFor: 'creation'!
new
    ^self error: 'Use new:base:'
!
new: ignore
    ^self new
!
new: size base: b
    ^(super new: size) init: b
!!
```

```
!RangedArray methodsFor: 'init'!
init: b
        base := (b - 1).      "- 1 because basicAt: works with a 1 base"
        ^self
!!
!RangedArray methodsFor: 'basic'!
rangeCheck: i
        ((i <= base) | (i > (base + (self basicSize)))) ifTrue: [
                'Bad index value: ' printOn: stderr.
                i printOn: stderr.
                (Character nl) printOn: stderr.
                ^self error: 'illegal index'
        ]
!
at: i
        self rangeCheck: i.
        ^self basicAt: (i-base)
!
at: i put: v
        self rangeCheck: i.
        ^self basicAt: (i-base) put: v
!!
```

The code has two parts; an initialization, which simply records what index you wish the array to start with, and the at: messages, which adjust the requested index so that the underlying storage receives its 1-based index instead. We've included a range check much like CheckedArray; its utility will demonstrate itself in a moment:

```
Smalltalk at: #a put: (RangedArray new: 10 base: 5) !
a at: 5 put: 0 !
a at: 4 put: 1 !
```

Since 4 is below our base of 5, a range check error occurs. But this check can catch more than just our own misbehavior!

```
a do: [:x| x printNl] !
```

Our do: message handling is broken! The stack backtrace pretty much tells the story:

```
RangedArray>>#rangeCheck:
RangedArray>>#at:
MethodContext>>#value:
Integer>>#to:by:do:
Integer>>#to:do:
RangedArray>>#do:
```

Our code received a do: message. We didn't define one, so we inherited the existing do: handling. We see that an Integer loop was constructed, that a code block was invoked, and that our own at: code was invoked. When we range checked, we trapped an illegal index. Just by coincidence, this version of our range checking code also dumps the index. We see that do: has assumed that all arrays start at 1.

The immediate fix is obvious; we implement our own do:

```
!RangedArray methodsFor: 'basic'!
do: aBlock
    1 to: (self basicSize) do: [:x|
        aBlock value: (self basicAt: x)
    ]
!!
```

But the issues start to run deep. If our parent class believed that it knew enough to assume a starting index of 1, why didn't it also assume that *it* could call basicAt:? Object-oriented methodology says that one object should be entirely opaque to another. But what sort of privacy should there be between a higher class and its subclasses? How many assumption can a subclass make about its superclass, and how many can the superclass make before it begins infringing on the sovereignty of its subclasses? There are rarely easy answers.

11.5. Basic Allocation

In this chapter, we've seen the fundamental mechanisms used to allocate and index storage. When the storage need not be accessed with peak efficiency, you can use the existing array classes. When every access counts, having the storage be an integral part of your own object allows for the quickest access. When you move into this area of object development, inheritance and polymorphism become trickier; each level must coordinate its use of the underlying array with other levels.

12. Further Studies

The question is always how far to go in one document. At this point, you know how to create classes. You know how to use inheritance, polymorphism, and the basic storage management mechanisms of Smalltalk. You've also seen a sampling of Smalltalk's powerful classes. The rest of this chapter simply points out areas for further study; perhaps a newer version of this document might cover these in further chapters.

12.1. Viewing the Smalltalk Source Code

Depending on the thoroughness of the person who installed GNU Smalltalk, it's possible to view the source code for a system method. For instance, to see how a Dictionary processes a do: message:

```
Dictionary edit: #do: !
```

The viewer is hard-coded as emacs; this may or may not work at your installation.

12.2. Other Ways to Collect Objects

We've seen Array, ByteArray, Dictionary, Set, and the various streams. You'll want to look at the Bag, LinkedList, and SortedCollection classes. For special purposes you'll want to examine ByteMemory and WordMemory.

12.3. Flow of Control

GNU Smalltalk has rudimentary support for *threads* of execution. The state is embodied in a Process class object; you'll also want to look at the ProcessorScheduler class.

12.4. Smalltalk Virtual Machine

GNU Smalltalk is implemented as a virtual instruction set. By invoking GNU Smalltalk with the -d option, you can view the byte opcodes which are generated as files on the command line are loaded. Similarly, running GNU Smalltalk with -e will trace the execution of instructions in your methods.

You can look at the GNU Smalltalk source to gain more information on the instruction set. A better first step if you want to pursue this subject is to start with "A Little Smalltalk"" by Tim Budd. The source code is freely available, and the book provides a solid introduction to Smalltalk-type virtual machines. The canonical book is from the original designers of Smalltalk:

> Smalltalk-80: The Language and its Implementation
> - Adele Goldberg and David Robson

12.5. Two Flavors of Equality

As first seen in chapter two, Smalltalk keys its dictionary with things like "#word", whereas we generally use "'word'". The former, as it turns out, is from class Symbol. The latter is from class String. What's the real different between a Symbol and a String? To answer the question, we'll use an analogy from C.

In C, if you have a function for comparing strings, you might try to write it:

```
strcpy(char *p, char *q)
{
        return (p == q);
}
```

But clearly this is wrong! The reason is that you can have two copies of a string—each with the same contents—but each at its own address. A correct string compare must walk its way through the strings and compare each element.

In Smalltalk, exactly the same issue exists, although the details of manipulating storage addresses are hidden. If we have two Smalltalk strings, both with the same contents, we don't necessarily know if they're at the same storage address. In Smalltalk terms, we don't know if they're the *same object.*

The Smalltalk dictionary is searched frequently. To speed the search, it would be nice to not have to compare the characters of each element, but only compare the address itself. To do this, you need to have a guarantee that all strings with the same contents are the same object. The String class, created like:

```
y := 'Hello' !
```

does not satisfy this. Each time you execute this line, you may well get a new object. But a very similar class, Symbol, will always return the same object:

```
y := #Hello !
```

In general, you can use strings for almost all your tasks. If you ever get into a performance-critical function which looks up strings, you can switch to Symbol. It takes longer to create a Symbol, and the memory for a Symbol is never freed (since the class has to keep tabs on it indefinitely to guarantee it continues to return the same object). You can use it, but use it with care.

12.6. Checking for the Two Types of Equality

This paper has generally used the strcmp()-ish kind of checks for equality. If you ever need to ask the question "is this the same object?", you use the "==" operator instead of "=":

```
Smalltalk at: #x put: 0 !
Smalltalk at: #y put: 0 !
x := 'Hello' !
y := 'Hello' !
(x = y) printNl !
(x == y) printNl !
x := #Hello !
y := #Hello !
(x = y) printNl !
(x == y) printNl !
```

Using C terms, the former compares contents like strcmp(). The latter compares storage addresses, like a pointer comparison.

12.7. Where to get Help

The newsgroup comp.lang.smalltalk is read by many people with a great deal of Smalltalk experience. There are several commercial Smalltalk implementations; you can buy support for these, though it isn't cheap. For the GNU Smalltalk system in particular, you can try the newsgroup gnu.smalltalk.bug. If all else fails, you can try the author at:

jtk@netcom.com

No guarantees, but the author will do his best!

12.8. Acknowledgments

Thanks to Steve Byrne for writing GNU Smalltalk in the first place. Great thanks to Mark Bush and Bob Roos for their meticulous jobs of proofreading this document, and the generous amounts of input they provided on refinements to the contents and structure. Thanks also to Andrew Berg for his comments on the early chapters of the document.

Any remaining errors are purely the fault of the author. This document is provided as-is, without warranty, but I will happily accept reports of any errors. If time permits, I will perhaps even release a corrected revision of the document.

Andy Valencia
325 Union Ave #359
Campbell, CA 95008
jtk@netcom.com
November 27, 1992

APPENDIX A
A Simple Overview of Smalltalk Syntax

Smalltalk's power comes from its treatment of objects. In this document, we've mostly avoided the issue of syntax by using strictly parenthesized expressions as needed. When this leads to code which is hard to read due to the density of parentheses, a knowledge of Smalltalk's syntax can let you simplify expressions. In general, if it was hard for you to tell how an expression would parse, it will be hard for the next person, too.

The following presentation presents the grammar a couple of related elements at a time. We use a BNF style of grammar, with some extensions. The form:

```
[ ... ]
```

means that "..." can occur zero or one times.

```
[ ... ]*
```

means zero or more;

```
[ ... ]+
```

means one or more.

```
... | ... [ | ... ]*
```

means that one of the variants must be chosen. Characters in double quotes refer to the literal characters. Most elements may be separated by white space; where this is not legal, the elements are presented without white space between them.

```
methods: "!" id ["class"] "methodsFor:" string "!" [method "!"] "!"
```

Methods are introduced by first naming a class (the *id* element), specifying "class" if you're adding class methods instead of instance methods, and sending a string argument to the methodsFor: message. Each method is terminated with an "!"; two "!"'s in a row signify the end of the new methods.

```
method: message [prim] [temps] exprs
message: id | binsel id | [keysel id]+
prim: "<" "primitive:" number ">"
temps: "|" [id]* "|"
```

A method definition starts out with a kind of template. The message to be handled is specified with the message names spelled out and identifiers in the place of arguments. A special kind of definition is the *primitive;* it has not been covered in this paper; it provides an interface to the underlying Smalltalk virtual machine. *temps* is the declaration of local variables. Finally, *exprs* (covered soon) is the actual code for implementing the method.

```
unit: id | literal | block | "(" expr ")"
unaryexpr: unit [ id ]+
primary: unit | unaryexpr
```

These are the "building blocks" of Smalltalk expressions. A *unit* represents a single Smalltalk value, with the highest syntactic precedence. A *unaryexpr* is simply a *unit* which receives a number of unary messages. A *unaryexpr* has the next highest precedence. A *primary* is simply a convenient left-hand-side name for one of the above.

```
exprs: [expr "."]* [["^"] expr]
expr: [id ":="]* expr2
expr2: primary | msgexpr [ ";" cascade ]*
```

A sequence of expressions is separated by "."'s and can end with a returned value ("^"). There can be leading assignments; unlike C, assignments apply only to simple variable names. An expression is either a primary (with highest precedence) or a more complex message. *cascade* does not apply to *primary* constructions, as they are too simple to require the construct--since all *primary* construct are unary, you can just add more unary messages:

```
1234 printNl printNl printNl !
```

```
msgexpr: unaryexpr | binexpr | keyexpr
```

A complex message is either a unary message (which we have already covered), a binary message ("+", "-", and so forth), or a keyword message ("at:", "new:", ...) Unary has the highest precedence, followed by binary, and keyword messages have the lowest precedence. Examine the two versions of the following messages. The second have had parentheses added to show the default precedence.

```
myvar at: 2 + 3 put: 4
mybool ifTrue: [ ^ 2 / 4 roundup ]
```

```
(myvar at: (2 + 3) put: (4))
(mybool ifTrue: ([ ^ (2 / (4 roundup)) ]))
```

```
cascade: id | binmsg | keymsg
```

A *cascade* is used to direct further messages to the same object which was last used. The three types of messages (*id* is how you send a unary message) can thus be sent.

```
binexpr: primary binmsg [ binmsg ]*
binmsg: binsel primary
binsel: selchar[selchar]
```

A binary message is sent to an object, which *primary* has identified. Each binary message is a binary selector, constructed from one or two characters, and an argument which is also provided by a examples*primary.*Some

```
1 + 2 - 3 / 4
```

which parses as:

```
(((1 + 2) - 3) / 4)
```

```
keyexpr: keyexpr2 keymsg
keyexpr2: binexpr | primary
keymsg: [keysel keyw2]+
keysel: id":"
```

Keyword expressions are much like binary expressions, except that the selectors are made up of identifiers with a colon appended. Where the arguments to a binary function can only be from *primary,* the arguments to a keyword can be binary expressions or *primary* ones. This is because keywords have the lowest precedence.

```
block: "[" [[":" id]* "|" ] exprs "]"
```

A code block is square brackets around a collection of Smalltalk expressions. The leading ": id" part is for block arguments.

```
literal: number | string | charconst | symconst | arrayconst
arrayconst: "#" array
array: "(" [number | string | symbol | array | charconst]* ")"
number: [[dig]+ "r"] ["-"] [hexDig]+ ["." [hexDig]+] ["e"["-"][dig]+].
string: "'"[char]*"'"
charconst: "$"char
symconst: "$"symbol
```

We have already shown the use of many of these constants. Although not covered in this paper, numbers can have a base specified at their front, and a trailing scientific notation. We have seen examples of character, string, and symbol constants. Array constants are simple enough; they would look like:

```
Smalltalk at: #a put: #(1 2 'Hi' $x $Hello 4 5) !
```

```
symbol: id | binsel | keysel[keysel]*
```

Symbols are mostly used to represent the names of methods. Thus, they can hold simple identifiers, binary selectors, and keyword selectors:

```
$hello
$+
$at:put:
```

```
id: letter[letter|dig]*
selchar: "+" | "-" | "*" | "/" | "~" | "|" | "," |
        "<" | ">" | "=" | "&"
hexdig: "0".."9" | "A".."F"
dig: "0".."9"
```

These are the categories of characters and how they are combined at the most basic level. *selchar* simply lists the characters which can be combined to name a binary message.