CS442
Module 1: Languages: Smalltalk

University of Waterloo

Winter 2025

1 Smalltalk

Let's start by getting the shock out of the way. Smalltalk is an object-oriented programming language. Indeed, Smalltalk is so object oriented that it's barely a procedural imperative language: Smalltalk's syntax doesn't even have conditionals or loops! How is that possible? Well, you don't need loops in the syntax of your language if block (lambda) objects have a `whileTrue` method which runs the block repeatedly, and you don't need conditionals if your boolean objects have an `ifTrue` method that evaluates its argument only if the boolean was true. If you're familiar with object-oriented languages that have a less pure approach to object-oriented programming (that is, nearly all of them), then Smalltalk's style will come as a bit of a shock. But, it doesn't take much getting used to, and before you've grown accustomed to it, you can simply mentally rewrite the conditions and loops you're comfortable with into Smalltalk's style.

Smalltalk's oddness doesn't stop there, however. Smalltalk is a class-based object-oriented language, as are C++, Java, Python, etc. You might then assume that a reasonable place to start with such a language is "what is the syntax to define a class?" Except... there isn't one. The Smalltalk standard does not include syntax for classes. Smalltalk was, at its inception, not just a programming language, but an operating system and development environment, and the way that you create a class is through the graphical workspace. You then add methods to the class through user-interface interactions, and then finally, the *body* of a method has a defined syntax. In order to reduce the semantic leap from languages that you may be more familiar with (and, quite frankly, to make it more gradeable), in this course, we'll be using an unusual variant of Smalltalk that behaves more like a conventional programming language, GNU Smalltalk. GNU Smalltalk accepts Smalltalk files and has an interactive REPL, like other programming languages. If you're interested in Smalltalk, you'll probably want to explore some other Smalltalk environments, such as Pharo (https://pharo.org/) and Squeak (https://squeak.org/), but bear in mind that only the syntax for the bodies of methods will look familiar.

This module assumes that you're familiar with some modern object-oriented languages such as C++ and perhaps Java, and that you're familiar with programming in general, and focuses on what will be unfamiliar in Smalltalk with that background.

2 My First Smalltalk Program

GNU Smalltalk on `linux.student.cs.uwaterloo.ca` is provided by CS442, in `/u/cs442/students`. To add it to your environment, use `source /u/cs442/students/env.sh`, or simply add `/u/cs442/students/inst/bin` to your `PATH`. Then run the command `gst`:

```
$ gst

GNU Smalltalk ready

st>
```

You may also want to install GNU Smalltalk on your own system. In most systems with package managers, it's in a package named `gnu-smalltalk`. Or, you can get the source at https://www.gnu.org/software/smalltalk/ and build it yourself. You should get the latest "alpha" version, which at the time of writing this is 3.2.91. Unfortunately, the latest release has a bug printing floating-point numbers; see the discussion in Section 8 on `floatfix.st`. This bug has been fixed in the version in `/u/cs442`. Note that the student servers sometimes have `/usr/bin/gst` as well, but you should make sure to use the version in `/u/cs442` because of this bug.

From here, we can write everyone's favorite first program:

```
st> 'Hello, world!' displayNl
Hello, world!
'Hello, world!'
st>
```

That... kind of worked? It actually worked fine: the first line is the behavior of the `displayNl` method itself, and the second line is the *return* from the `displayNl` method. `displayNl` returns the thing that it displayed.

Let's take a moment to examine *how* this worked, though, because it probably looks unfamiliar. Our command, `'Hello, world!' displayNl`, has two components: the string and the message. The string should look familiar enough, but it's worth noting that in Smalltalk, strings can *only* be delimited by single quotes (`'`), not double quotes (`"`). In addition, strings in Smalltalk are objects of the class `String`.

In fact, *everything* in Smalltalk is an object! Yes, even the class `String` itself is an object, of the class `Class`! And, of course, the class `Class` is itself an object, of, you guessed it, the class `Class`.

The second part is the message. This command sends the `displayNl` message to a string object. When an object receives a message (in this case, `displayNl`), it looks up a corresponding method in its class. String objects are of the class `String`, so `String`'s `displayNl` method is invoked (sometimes written `String>>displayNl` to clarify which class's method is meant), and the result of the expression (`'Hello, world!' displayNl`) is the return value of the method. This whole process should feel familiar if you've used any object-oriented language, so rather than describing the whole process of sending a message and responding by invoking a method every time, we will simply describe this process as "calling the `displayNl` method"[1].

Smalltalk is quite purely object oriented: there are no procedures that aren't methods, and there is no control flow but calling methods and returning. So, if methods are all that there is, then the way to display a string is to call the display method on the string—i.e., ask the string to display itself. If C++ supported this style of displaying strings, it might look something like this:

```
("Hello, world!").displayNl();
```

The parentheses and dots and other syntactic clutter are there because C++ has so many other ways of expressing yourself. When all you can do is call methods, you don't need a dot or parentheses to say that you're calling a method. Of course you're calling a method; that's all you can do! So, all that extra syntax falls away, and the simple way to call a method is just to name it: `'Hello, world!' displayNl`.

Since single quotes delimit strings, if you want to put a single quote in a string, you need to escape it (specify that this single quote is not the end of the string). To do so, simply use two single quotes instead of one:

```
st> 'This isn''t SUCH a bad programming language!' displayNl.
This isn't SUCH a bad programming language!
'This isn''t SUCH a bad programming language!'
st>
```

Note that the string it actually printed had only one single quote. The second line is showing what value was returned by `displayNl`, and to show that a string was returned, it surrounds it in quotes, and escapes the quote again.

You can use Ctrl+D or the extremely memorable command `ObjectMemory quit` to leave the GNU Smalltalk interactive shell. We'll mostly be using Smalltalk files, since Smalltalk was never really meant to be used in a read-eval-print-loop (REPL) like this.

Aside: The `Nl` in `displayNl` means "newline". If you didn't want to print with a newline at the end, just use `display`!

[1] Please don't tell the Smalltalk purists I said "calling a method". They would have me pilloried for using that terminology.

3 Let's do Math

Create a file named `math.st` (or whatever you'd like) in your favorite text editor. Let's write a program to do some basic arithmetic:

```
30 + 12 displayNl
```

And run it:

```
$ gst math.st
12
```

That result was probably not what you expected. The reason is that everything in Smalltalk is an object, so all behaviors are method calls, even `+`. It should be clearer if we add some parentheses to show the precedence:

```
30 + (12 displayNl)
```

A zero-argument method call like `displayNl` binds more tightly (has higher precedence) than a binary method call like `+`, so it came first. Let's rewrite this in C++-like method-call syntax to show every step:

```
(30).operator+((12).displayNl());
```

The `displayNl` method was applied on the `12`, and then its *result* was passed to the `+` method on `30`. On the interactive shell, we would have also displayed the result of the whole computation, which is 42, but now that we're writing actual Smalltalk files, the only output that's displayed is what you explicitly ask for. We can fix this with parentheses (which, contrary to many other languages, are *never* method calls). We can also clarify our code with a comment, which in Smalltalk is surrounded by double quotes:

Program	Output
`" Display the sum of 30 and 12, rather than adding the display of 12 to 30 "` `(30 + 12) displayNl`	`42`

OK! By putting `(30 + 12)` in parentheses, we're forcing it to be evaluated first. Then, the `displayNl` method is called on the result, `42`. Now, let's do some very slightly more sophisticated math:

Program	Output
`(30 + 6 * 2) displayNl`	`72`

Once again, a surprising result. If we apply the usual rules of math, then * has higher precedence than `+`, so we would perform `6 * 2`, then `30 + 12`. But, this isn't (just) math, and `+` and * aren't just operators, they're methods. Smalltalk doesn't actually know the rules of mathematical precedence, it just knows how to call methods on objects. In this code, it just knows to call the `+` method because you asked it to, and then call the * method because you asked it to. In fact, Smalltalk doesn't even know what the mathematical operators are; if you want to use `@` as an operator for your own class, you just need to name a method in that class "`@`".

Since it knows nothing of the order of operations, Smalltalk treats all binary operators with equal precedence, going left to right. We can now fix our simple math with even more parentheses:

Program	Output
`" Multiply first "` `(30 + (6 * 2)) displayNl`	`42`

Since `displayNl` returns the object being displayed, we can actually observe this left-to-right behavior quite nicely, by printing the result of every intermediate calculation:

Program	Output

`(30 displayNl + (6 displayNl * 2 displayNl) displayNl) displayNl`	`30`
	`6`
	`2`
	`12`
	`42`

Exercise 1. Work through the above Smalltalk program and why it outputs exactly what it outputs.

We can perform multiple statements by separating them with a dot (`.`). In essence, you can think of the dot like a semicolon (`;`) in C and languages that follow its style:

Program	Output

`(30 + (6 * 2)) displayNl.`
`(10 - 15) displayNl.`
`(10 * 10) displayNl.`
	`42`
	`-5`
	`100`

It is not necessary to end the last statement in a list of statements with a dot, but it is common to do so.

Video 1.s.1 (https://student.cs.uwaterloo.ca/~cs442/W25/videos/1.s.1/): Basic math

4 Booleans, Conditions, and Loops

It should come as no surprise that Smalltalk has two boolean values: `true` and `false`. By this point, it should hopefully also not come as a surprise that `true` and `false` are objects of the class `Boolean`, and they have some useful methods.

In most modern programming languages, using a boolean to conditionalize execution looks something like this:

```
if (condition) {
    statements;
}
```

In Smalltalk, there are no conditional statements like this in the syntax. Instead, just like everything else, we do things conditionally by calling a method. In this case, the `isTrue:` method on `Boolean`s:

Program	Output
`true ifTrue: ['true is true!' displayNl]`	`true is true!`

This example has introduced two new elements of Smalltalk syntax. First, the simple one. Note the `:` on `ifTrue:`. Methods named with a colon like this expect an argument. Of course, in this case, the interesting part is what that argument *is*.

Now, the more interesting new element. The expression `['true is true!' displayNl]` defines a *block*. A block is sort of like an anonymous function, in that it is a packaging of code which can be evaluated later, but this analogy is imperfect, so we'll simply use the standard Smalltalk term, "block". We pass that block as an argument to the `ifTrue:` method. Like an anonymous function, `ifTrue:` can then choose to call it, or not to call it.

You may think that the implementation of `ifTrue:` has to "cheat" and do a conventional `if` statement to work. In fact, its implementation is far more object-oriented than that. The object `true` is actually of the class `True`, which is a *subclass* of the class `Boolean`. Some methods are implemented on `Boolean`, and those can be used on `true`: like in other class-based languages, subclasses inherit the behavior of their superclasses. But, the method `ifTrue:` is implemented on `True`, and all it does is evaluate the block passed as its argument. It doesn't need to check if the boolean is `true`. It knows it's `true` because this is a method on the `True` class, and only `true` is of the class `True`. Conversely, the method `ifTrue:` on the `False` class does nothing.

A block boxes up code, and it only *runs* the code if someone asks for it. It gives us a way of making it optional to execute the code at all.

Since both `True` and `False` support the `ifTrue:` method, but `ifTrue:` only evaluates its argument on `True`, this allows us to conditionalize the execution of a block, with nothing but objects and methods!

Let's put this together with our math from before, adding the operators for comparing numbers:

Program

```
" Mathematical sanity checks "
40 < 42 ifTrue: [
    '40 is less than 42' displayNl].
40 < 24 ifTrue: [
    '40 is less than 24???' displayNl].
10 + 10 > 19 ifTrue: [
    '10 + 10 is greater than 19'
    displayNl].
50 >= 50 ifTrue: [
    '50 is greater than or equal to 50'
    displayNl].
10 + 10 = 20 ifTrue: [
    '10 + 10 is 20' displayNl].
10 + 10 ~= 30 ifTrue: [
    '10 + 10 is not 30' displayNl].
```

Output

```
40 is less than 42
10 + 10 is greater than 19
50 is greater than or equal to 50
10 + 10 is 20
10 + 10 is not 30
```

Equality and inequality work on all values, but the rest are specific to numbers. Remember when using these operators that they simply evaluate left-to-right. There is still no operator precedence:

Program

```
10 + 10 > 10 + 9 ifTrue: [
    'This looks like fine math to me'
    displayNl].
```

Output

```
Object: true error: did not understand #+
MessageNotUnderstood(Exception)>>signal (
    ExcHandling.st:254)
True(Object)>>doesNotUnderstand: #+ (
    SysExcept.st:1448)
UndefinedObject>>executeStatements (
    example.st:1)
```

These error messages can be a bit dense, but only the first line really matters: `true` doesn't know how to `+`, and because Smalltalk evaluates left-to-right, we tried to perform `+ 9` on the `true` that was a result of `10 + 10 > 10`. As usual, we can fix this with more parentheses:

Program

```
10 + 10 > (10 + 9) ifTrue: [
    '20 is greater than 19' displayNl].
```

Output

```
20 is greater than 19
```

You can build a lot out of `ifTrue:`, but it doesn't have an "else" branch. Booleans also have two other methods, `ifFalse:` and `ifTrue:ifFalse:`, which cover other cases:

Program

```
100 > 10 ifFalse: [
    '100 isn''t greater than 10???'
    displayNl].

" There is no universally-accepted style
    for how to indent ifTrue:ifFalse:.
    Just be consistent in your own code.
    "
10 >= 10
    ifTrue: [
        '10 is greater than or equal to 1
    0' displayNl
    ] ifFalse: [
        '10 isn''t greater than or equal
    to 10???' displayNl
    ].
```

Output

```
10 is greater than or equal to 10
```

These hopefully behave as you would expect them to, but note how the call to `ifTrue:ifFalse:` *looks*. First is the operator which generates the `true` on which we'll be calling the method. Then comes `ifTrue:`, *then* comes the true argument value (a block), *then* comes `ifFalse:`, and *then* comes the false argument value (another block). This style of method call syntax is Smalltalk's second greatest difference from more popular languages. Because parameter names and argument values can be intermixed in this way, it's very important to make sure you put a dot at the end of your statements. If you don't, the error messages can be extremely confusing.

Video 1.s.2 (https://student.cs.uwaterloo.ca/~cs442/W25/videos/1.s.2/): ifTrue:ifFalse:

Generally speaking, Smalltalk programmers try to name their methods such that, with the actual argument values in place, it reads a bit like an English sentence. For instance, on arrays (which we will look at in more detail later), the method to put a value into the array at a given location is `at:put:`. This would be written *my-array* `at:` *the-position* `put:` *my-value*. This reads as "in my array *at* the position *put* my value".

Now we've done conditions, but not loops. The simplest way to loop in Smalltalk is over a numerical range:

Program

```
1 to: 10 do: [:x|
    x displayNl
].
```

Output

```
1
2
3
4
5
6
7
8
9
10
```

Note our new addition to blocks. Like methods, blocks can take arguments. Block arguments are labeled with `:` at the beginning of the block, and the list of block arguments is ended with `|`.

Of course, this can be mixed with math:

Program

```
1 to: 2*2 do: [:x|
    x displayNl
]
```

Output

```
1
2
3
4
```

What if I need to do a more sophisticated loop? Like any other programming language, it's possible to loop while some condition is true (or while some condition is false), but to learn how to do that, first we'll have to introduce variables, so our condition can actually change.

5 Variables and Assignment

To declare variables, surround them in pipes (`|`). To assign to them, use `:=`.

Program

```
| x y |
1 to: 4 do: [:z|
    x := z * 2.
    y := x * 2.
    y displayNl.
].
```

Output

```
4
8
12
16
```

Of course, this isn't an especially intersting use of variables. Let's use variables and math to make a more interesting loop:

Program

```
| x |
x := 1.
[ x < 64 ] whileTrue: [
    x displayNl.
    x := x * 2.
].
```

Output

```
1
2
4
8
16
32
```

One major difference from our conditionals (`ifTrue:ifFalse:`) is worth noting: our condition itself is in a block! Why did we need it in a block here, and why *didn't* we need it in a block before? The answer is simple, but not obvious: anything

that needs to be run repeatedly needs to be in a block. In order to check whether `x < 64` multiple times, we need to put it in a block, so that block can be made to run repeatedly, re-checking whether `x < 64` every time around the loop. Of course, the block given as an argument to `whileTrue:` is also evaluated multiple times! There's also a `whileFalse:`, which behaves as you'd expect.

Video 1.s.3 (https://student.cs.uwaterloo.ca/~cs442/W25/videos/1.s.3/): Loops

Exercise 2. Using the loops we've seen here, write a program to output the Fibonacci sequence.

6 Containers

Smalltalk has various standard containers, including lists, arrays, strings (which are arrays of characters), and dictionaries.

6.1 Lists

Although various Smalltalk classes are implemented as lists, the most frequently useful list type is `OrderedCollection`. Until now, we've only created objects implicitly—numbers are objects, blocks are objects—but never explicitly built an object from a class. As with everything, the way to build an object from a class in Smalltalk is to call a method on the class; usually, the `new` method. Thus, we create an `OrderedCollection` with `OrderedCollection new`:

Program

```
| lst |
lst := OrderedCollection new.
lst displayNl.
```

Output

```
OrderedCollection ()
```

The display format for an `OrderedCollection` is simply `OrderedCollection` followed by its elements in parentheses.

Add elements to an `OrderedCollection` with `add:`:

Program

```
| lst |
lst := OrderedCollection new.
lst add: 42.
lst displayNl.
```

Output

```
OrderedCollection (42 )
```

You can also use `addFirst:` to add to the beginning of the list instead of the end. If you have another collection (whether an `OrderedCollection` or not), you can add *all* of its elements with `addAll:`. You can concatenate two lists into a new list with the `,` operator:

Program

```
| lsta lstb lstc |
lsta := OrderedCollection new.
lsta add: 1.
lstb := OrderedCollection new.
lstb add: 3.
lstc := lsta , lstb.
lstc add: 9.
lsta displayNl.
lstb displayNl.
lstc displayNl.
```

Output

```
OrderedCollection (1 )
OrderedCollection (3 )
OrderedCollection (1 3 9 )
```

Alternatively, you can use `with:` (or `with:with:`, etc) while building the `OrderedCollection` to create a list and add elements in one step. Loop over the list with `do:`:

Program

```
| lst |
lst := OrderedCollection
    with: 42
    with: 12.
lst do: [:v|
    (v < 20) displayNl.
].
```

Output

```
false
true
```

You can access members of the list directly with `at:`, and get the number of elements in the list with `size`:

Program

```
| lst i |
lst := OrderedCollection with: 100 with:
    9.
i := 1.
[ i <= lst size ] whileTrue: [
    (lst at: i) displayNl.
    i := i + 1.
].
lst at: i.
```

Output

```
100
9
Object: OrderedCollection new: 16 "<0x7f2
    ae1c2b3b0>" error: Invalid index 3:
    index out of range
SystemExceptions.IndexOutOfRange(
    Exception)>>signal (ExcHandling.st:25
    4)
SystemExceptions.IndexOutOfRange class>>
    signalOn:withIndex: (SysExcept.st:660
    )
OrderedCollection>>at: (OrderColl.st:100)
UndefinedObject>>executeStatements (
    example.st:8)
```

Note that lists—and all other collections in Smalltalk—are 1-indexed, not 0-indexed, so the first element is `at: 1`. In this example, our error was using `at: i` after the loop, since at that point, `i > lst size`. As before, the first line tells us what we need to know: "Invalid index 3: index out of range".

You can also *change* an element in an `OrderedCollection` with `at:put:`:

Program

```
| lst i |
lst := OrderedCollection with: 0 with: 0.
i := 1.
[ i <= lst size ] whileTrue: [
    lst at: i put: i*2.
    i := i + 1.
].
lst displayNl.
```

Output

```
OrderedCollection (2 4 )
```

Finally, you can remove the first or last element of a list with `removeFirst` or `removeLast` respectively. These methods return the removed element, so are useful for using the list as a queue or stack.

Lists are *lists*, so indexing with `at:` is slow, but adding elements with `add:` is fast.

6.2 Arrays

Arrays are of a fixed size, and their size must be declared when they are created. As such, rather than a simple `new` method, Arrays have `new:`, which takes the size as an argument:

Program

```
| arr i |
arr := Array new: 10.
arr at: 1 put: 10.
arr displayNl.
i := 1.
[ i <= arr size ] whileTrue: [
    arr at: i put: i-1.
    i := i + 1.
].
arr displayNl.
```

Output

```
(10 nil nil nil nil nil nil nil nil nil )
(0 1 2 3 4 5 6 7 8 9 )
```

Note that before we've put a value into a slot of the array, the value is `nil`. `nil` is similar to `null`, `nullptr`, or `NULL` in other programming languages. But, `nil` is a fully-fledged object with methods and a class. You can check if a value is `nil` with a normal comparison, e.g. `x = nil`, but it is more common to use the `isNil` method, e.g. `x isNil`, which returns `true` only for `nil`.

Arrays can also be created simply with array literals, which are written with braces, with the elements separated by dots:

Program

```
| arr |
arr := {2. 4. 6. 8. 10}.
arr do: [:v|
    (v / 2) displayNl.
].
```

Output

```
1
2
3
4
5
```

Because their size is fixed, Arrays don't support `add:` or its variants. However, every other `OrderedCollection` method shown in the previous section behaves the same as in `OrderedCollection`s, including concatenation. Arrays are arrays, so indexing and mutating with `at:` is fast, but adding elements is so slow that the language has simply prevented it!

It is common to convert back and forth between `OrderedCollection`s and Arrays to benefit from each kind's advantages when needed. All collections have an `asOrderedCollection` method to convert to an `OrderedCollection`, and an `asArray` method to convert to an array:

Program

```
| x |
x := {10. 10. 30} asOrderedCollection.
x add: 40.
x := x asArray.
x at: 2 put: 20.
x displayNl.
```

Output

```
(10 20 30 40 )
```

Aside: Array is actually a subclass of `OrderedCollection` which replaces its internal data structure with an array instead of a list. `OrderedCollection` is the generic superclass of all ordered collections, but is also a fully implemented list class itself. This is why `OrderedCollection` has a strange, generic-sounding name, while Array has a very precise, descriptive name.

6.3 Strings and Characters

We've already seen strings: they're delimited by single quotation marks (`'`). In fact, strings are just a special kind of read-only `Array`, an `Array` of `Character`s. Anything you can do with an `Array`, you can do with a `String`, except that you cannot change it:

Program

```
| s |
s := 'Hello, world!'.
s do: [:c|
    c displayNl.
].
```

Output

```
H
e
l
l
o
,

w
o
r
l
d
!
```

We can convert a `String` to an `Array` and vice-versa, usually in order to make it modifiable. To specify an element of the `Character` type, prefix the character with a dollar sign ($):

Program

```
| s |
s := 'Hello, world!' asArray.
s displayNl.
s at: 13 put: $?.
s := s asString.
s displayNl.
```

Output

```
($H $e $l $l $o $, $  $w $o $r $l $d
    $! )
Hello, world?
```

Any character after a $ will be taken as a character literal, so, for instance, you can get a literal newline like so:

Program

```
| c |
c := $
.
'There will be two blank lines here:'
    displayNl.
c displayNl. " One blank line from c,
    the other from displayNl "
'^^^' displayNl.
```

Output

```
There will be two blank lines here:


^^^
```

Using $ like this is usually considered to be in poor form, however, so the `Character` class also has a constructor that directly creates a newline character, `lf`:

Program

```
| c |
c := Character lf.
'There will be two blank lines here:'
    displayNl.
c displayNl. " One blank line from x, the
     other from displayNl "
'^^^' displayNl.
```

Output

```
There will be two blank lines here:


^^^
```

Video 1.s.4 (https://student.cs.uwaterloo.ca/~cs442/W25/videos/1.s.4/): Ordered collections

6.4 Dictionaries

Smalltalk also supports key-value maps, called `Dictionary`s. Any value can be used as a key, and a single `Dictionary` can contain keys of multiple types, though most dictionaries in practice only use one.

Create a `Dictionary` with `new`, and add key-value associations with `at:put:`, like other collections:

Program	Output

`| dict |`
`dict := Dictionary new.`
`dict at: 'Hello' put: 'world'.`
`(dict at: 'Hello') displayNl.`
	`world`

In addition, you can check whether a key is present with `includesKey:`, and remove a key-value pair with `removeKey`:

Program	Output

`| dict |`
`dict := Dictionary new.`
`dict at: 'Hello' put: 'world'.`
`[ dict includesKey: 'Hello' ] whileTrue: [`
`(dict at: 'Hello') displayNl.`
`dict removeKey: 'Hello'.`
`].`
	`world`

You can loop over `Dictionary`s with `do:`, like other collections, but the block will only receive the *values*, not the keys. To loop over both keys *and* values, use `keysAndValuesDo:`, which takes a two-argument block (which we haven't seen before):

Program	Output

`| dict |`
`dict := Dictionary new.`
`dict at: 'Hello' put: 'world'.`
`dict at: 42 put: 12.`
`dict keysAndValuesDo: [:key :value|`
`'At key ' display.`
`key display.`
`' the dictionary has the value '`
`display.`
`value displayNl.`
`].`
	`At key Hello the dictionary has the value world`
	`At key 42 the dictionary has the value 12`

Note that in two-argument blocks, the space between an argument name and the next `:` is mandatory.

7 Creating Your Own Classes

Until this point, our focus has been on learning the syntax of Smalltalk and how to use its built-in types. Now, let's make our own type. We will make a "reverse Polish notation" calculator.

If you're not familiar with reverse Polish notation (RPN) calculators, they are a form of calculator which represents its input as a list of commands which operates on a stack. A number is a command to push that number to the stack, and an operator is a command to pop two values from the stack, perform the relevant operation, and then push the result onto the stack. For instance, the mathematical expression $1 + 2 * 3$ is written in RPN as `1 2 3 * +`, and $(1 + 2) * 3$ is written as `1 2 + 3 *`.

Naturally, our RPN calculator will need a stack. It will need a method to push a number onto the stack, and methods for each of the supported operators.

The first question is, how do we create a new class? Naturally, we call a method on an existing class! Namely, the `subclass:` method. As there's nothing in particular our RPN calculator should be a subclass of, we'll just make it a subclass of `Object`, the base class:

```
1 Object subclass: #RPNCalculator.
```

We can verify that our new class does in fact exist:

Program	Output
`1 Object subclass: #RPNCalculator.` `2 RPNCalculator new displayNl.`	`a RPNCalculator`

"a RPNCalculator" is simply the output of `Object>>display`, the general-purpose `display` method that works—albeit not usefully–on all objects. The odd `#` thing is called a "symbol", and is essentially just how you represent a string that's intended to be used as a name in Smalltalk. Other than this one use and in error messages, you're unlikely to ever need symbols.

It's a bit pointless to create a subclass with absolutely nothing in it, of course. You *can* go one-by-one through the desired methods of the class and add them all, but GNU Smalltalk offers a shorthand for defining a method in a somewhat more familiar style:

```
1 Object subclass: RPNCalculator [
2     | stack |
3
4     push: aNumber [
5         stack add: aNumber.
6         ^ aNumber
7     ]
8 ].
```

This program creates a class named `RPNCalculator` as a subclass of `Object` (and note that in this shorthand, no `#` is needed), makes instances of the `RPNCalculator` class have the field `stack` (actually called an "instance variable" in Smalltalk), and adds a method `push:` which adds its argument to the stack (assuming that the stack is simply an `OrderedCollection`). This snippet also shows how to return a value from a method: with a carat (`^`).

We haven't made our constructor yet, but to do so, we're going to have to understand one crucial detail about Smalltalk: Smalltalk instance variables (fields) are *always* private. There is no syntax to access an instance variable of another object. But, the constructor is a method of the *class*, not a method of objects *of* the class (think of `OrderedCollection new`), and so it can't access the instance variables even of the object it's *creating*. Because of this, it's common for constructors to be in two parts: the class has the constructor, and instances have an *initializer*. The initializer can access its own instance variables, so the constructor calls the initializer. We'll demonstrate this by adding a constructor for `RPNCalculator`:

```
1  Object subclass: RPNCalculator [
2      | stack |
3
4      RPNCalculator class >> new [
5          | r |
6          r := super new.
7          r init.
8          ^ r
9      ]
10
11     " Stack implemented as a list "
12     init [
13         stack := OrderedCollection new.
14     ]
15
16     push: aNumber [
17         stack add: aNumber.
18         ^ aNumber
19     ]
20 ].
```

There are a few things to note about this example:

- On line 4, we declare the `new` method. But, `new` is not a method on *instances* of `RPNCalculator`, but on the `RPNCalculator` class itself. The way we specify the `RPNCalculator` class is `RPNCalculator class`. The way we specify that we're implementing a method on that class rather than the class we're currently defining is `>>`. Thus, we define the method with `RPNCalculator class >> new`.

- Because we're overriding `new`, which is how you create a new object, we've actually lost the ability to *create* the object. But, we can call the method we overrode, `Object class >> new`, with `super`, as `super new`.
- The `new` method cannot access the `stack` variable of the object it just created, so instead it calls `init`.
- The `init` method on line 12 *can* access `stack`, so it creates the needed `OrderedCollection`.
- The `push:` method can now safely assume that `stack` is an `OrderedCollection`.

Now, we'd like to be able to do math. Let's start with addition:

```
1  Object subclass: RPNCalculator [
2  (...)
3      add [
4          | x y r |
5          y := stack removeLast.
6          x := stack removeLast.
7          r := x+y.
8          stack add: r.
9          ^ r
10     ]
11 ].
```

This method pops two elements from the stack with `removeLast`, then adds them, then pushes the result onto the stack, and returns the result. We can—and will—write methods for each of the usual operators, but they'll all be doing the same thing: pop two values, do a calculation with them, then push the result. Surely there's some way we can write this so that we don't write the same code over and over again? Of course there is: we need to use blocks! Blocks allow us to box up a calculation, like an anonymous function, and boxing up the actual calculation so we can separate it from the stack behavior is exactly what we want to do. We can write a generic "binary operator" method that takes a block as its argument, and expects the *block* to do the correct calculation. As yet, although we've passed blocks to other methods, we haven't actually evaluated blocks ourselves, so we'll have to learn how to do that, too.

At this point, you should be able to guess how you evaluate a block. You call a method on it, of course! Specifically, the `value` method if it's a zero-argument block, the `value:` method if it's a one-argument block, and the `value:value:` method if it's a two-argument block. Our block is supposed to perform a binary operator, so it'll be a two-argument block:

```
1  Object subclass: RPNCalculator [
2  (...)
3      binary: operatorBlock [
4          | x y r |
5          y := stack removeLast.
6          x := stack removeLast.
7          r := operatorBlock value: x value: y.
8          stack add: r.
9          ^ r
10     ]
11
12     add [
13         ^ self binary: [:x :y| x + y]
14     ]
15 ].
```

Our new implementation of `binary:` is almost exactly like our old implementation of `add`, but it takes an `operatorBlock` argument, and then calls that block on line 7 with `value:value:`. Our add method is now simplified to one line: call the `binary:` method on `self` (the Smalltalk equivalent of "`this`"), with a block as an argument which simply adds together its two arguments. The result of `self binary:` is returned from `add`.

Now, let's add the remaining methods for other mathematical operations, and finish our RPN calculator:

```
Object subclass: RPNCalculator [
    | stack |

    RPNCalculator class >> new [
        | r |
        r := super new.
        r init.
        ^ r
    ]

    " Stack implemented as a list "
    init [
        stack := OrderedCollection new.
    ]

    push: aNumber [
        stack add: aNumber.
        ^ aNumber
    ]

    binary: operatorBlock [
        | x y r |
        y := stack removeLast.
        x := stack removeLast.
        r := operatorBlock value: x value: y.
        stack add: r.
        ^ r
    ]

    add [
        ^ self binary: [:x :y| x + y]
    ]

    sub [
        ^ self binary: [:x :y| x - y]
    ]

    mul [
        ^ self binary: [:x :y| x * y]
    ]

    div [
        ^ self binary: [:x :y| x / y]
    ]
].
```

> **Aside:** It is generally considered good form in Smalltalk to have *many short methods*, rather than few long methods. That is, break methods down into their minimal useful components.

Now we have a few options for using and testing `RPNCalculator`. First, assuming we've stored this in a file named `rpncalc.st`, we can use it on the GNU Smalltalk REPL by running `gst` with the argument `-`:

```
$ gst rpncalc.st -
GNU Smalltalk ready

st> | x |
st> x := RPNCalculator new.
a RPNCalculator
st> x push: 1.
1
st> x push: 2.
2
st> x push: 3.
3
st> x mul.
6
```

```
st> x add.
7
st>
```

As you may guess from that command line, you can also write tests in a separate `.st` file, and run both with `gst rpncalc.st tests.st`. Finally, you can write more sophisticated tests using the built in testing framework SUnit (https://www.gnu.org/software/smalltalk/manual/html_node/SUnit.html).

Video 1.s.5 (https://student.cs.uwaterloo.ca/~cs442/W25/videos/1.s.5/): RPNCalculator

8 Subclasses

Our `RPNCalculator` is mostly ambivalent to the kind of number you pass in. GNU Smalltalk actually has several ways of storing numbers: integers, precise fractions, and floating-point numbers. Unfortunately, the different kinds of numbers don't always behave well when you mix and match, so if we want our calculator to be robust, we should make sure that incoming numbers are of a predictable type. We can do this by making subclasses of `RPNCalculator` which override `push:`, but convert to the desired type:

```
1  RPNCalculator subclass: FractionalRPNCalculator [
2      push: aNumber [
3          ^ super push: (aNumber asFraction)
4      ]
5  ].
6
7  RPNCalculator subclass: FloatRPNCalculator [
8      push: aNumber [
9          ^ super push: (aNumber asFloat)
10     ]
11 ].
```

Unsurprisingly, `asFraction` converts a number to a precise fraction, and `asFloat` converts a number to a float.

We can use one of these subclasses by calling `FractionalRPNCalculator new` or `FloatRPNCalculator new`. Note that we didn't need to re-implement `new` for each; they inherited their `new` from their superclasses. And, we didn't need to implement `binary:`, `add`, `sub`, `mul`, or `div`, for the same reason. If we needed to specialize one or all of these for these types, we could have overridden them, but all of these types do math the same, so we didn't need to.

Note that the latest release of GNU Smalltalk has an unfortunate bug in its code to display floating point numbers, and because of this, printing floating point numbers will often fail. This is easily worked around, though, by loading in the float printing code from a later version of GNU Smalltalk. This code is available on the course web site's assignments tab, named `floatfix.st`. Load it before your own code, such as `gst floatfix.st rpncalc.st`. This bug has been fixed in the version of GNU Smalltalk in `/u/cs442`.

9 Returning and Blocks

Earlier we said that blocks are similar but not identical to anonymous functions, and the major difference is how they interact with returning. Blocks themselves evaluate to a value—we used this in our `binary:` method—but this isn't done with `^`, which is how we return from methods. The value of a block is simply the value from the last statement in the block. On the other hand, when you return (with `^`) inside of a block, the *surrounding method* returns, not just the block! For instance, consider the output of this code:

Program

```
Object subclass: Funny [
    t: block [
        'I''m about to run the block!'
    displayNl.
        block value ifTrue: [
            ^true
        ].
        'The block''s value was false!'
    displayNl.
        ^false
    ]
]

| x |
x := Funny new.
x t: [true].
x t: [false].
```

Output

```
I'm about to run the block!
I'm about to run the block!
The block's value was false!
```

This is particularly useful for making chains of comparisons without getting too deeply nested, or for "switch"-like patterns:

```
runCommand: cmd [
    (cmd = 'push') ifTrue: [
        self push.
        ^self
    ].

    (cmd = 'pop') ifTrue: [
        self pop.
        ^self
    ].

    (cmd = 'rotate') ifTrue: [
        self rotate.
        ^self
    ].
]
```

10 Display

The `display` and `displayNl` methods use the `displayString` method to convert a value to a string. So if we want our `RPNCalculator` to display its stack when `displayNl` is called on it, we could extend it with a `displayString` method:

```
displayString [
    | r |
    r := 'RPNCalculator ('.
    stack do: [:x|
      r := r , (x displayString) , ' '.
    ].
    ^r , ')'
]
```

Note that this only affects `display` and `displayNl`, which are not how values are displayed in the interactive REPL. You can override `printString` for that.

11 Debugging

By default, GNU Smalltalk gives stack traces, but nothing else for debugging. However, you can get a debugger (the `DebugTools`) by loading it into your GNU Smalltalk environment along with your own code, and then you will automatically enter the debugger when there is a problem:

```
$ gst /u/cs442/students/inst/share/smalltalk/examples/MiniDebugger.st rpncalc.st -
Loading package DebugTools
GNU Smalltalk ready

st> | x |
st> x := RPNCalculator new.
a RPNCalculator
st> x push: 1.
1
st> x push: 0.
0
st> x div.
'1 error: The program attempted to divide a number by zero'
ZeroDivide(Exception)>>signal (ExcHandling.st:254)
SmallInteger(Number)>>zeroDivide (SysExcept.st:1426)
SmallInteger>>/ (SmallInt.st:277)
optimized [] in RPNCalculator>>div (tmp.st:42)
RPNCalculator>>binary: (tmp.st:24)
RPNCalculator>>div (tmp.st:42)
UndefinedObject>>executeStatements (a String:1)
      6         ^self activateHandler: (onDoBlock isNil and: [ self isResumable ])
(debug)
```

The `(debug)` prompt acts similarly to `gdb`: you can go up and down the callgraph, step through code, etc. Use `help` for a list of commands.

Rather than `print`, `MiniDebugger` has `i` (for "inspect"), which can inspect objects in great detail.

If you want to step through code easily, you'll have to cause it to break, so that it enters the debugger. This is most easily done by adding the statement `self halt` to your problematic code.

12 GNU Smalltalk in this Course

This course will ask you to use Smalltalk to implement interpreters for (very simple) programming languages. We will *not* ask you to parse code, and will instead provide parsers for you to use. You are free to ignore our provided parsers and write your own if you'd like, but there's no reason to do so. Abstract syntax trees will be built manually as Smalltalk objects, and examples given. In general, you should be writing the solution into one .st file, and tests in a separate file loaded afterward, and our input will also work in this way. For example, a simple test case for `rpncalc.st` might look like this:

```
| x |
x := RPNCalculator new.
x push: 10.
x push: 5.
x div = 2 ifFalse: [
    'Division fails!' displayNl.
].
```

And, if that's in the file `test.st`, would be run as `gst rpncalc.st test.st`.

You may use only the libraries built into GNU Smalltalk, `floatfix.st`, and your own code.

13 More Resources

The content described in this document should be sufficient for everything needed in this course. If you feel that more functionality would help you, you can look at the GNU Smalltalk Library Reference (https://www.gnu.org/software/smalltalk/manual-base/html_node/) and User's Guide (https://www.gnu.org/software/smalltalk/manual/html_node/index.html). If you're interested in Smalltalk more broadly, you will probably want to look into a more conventional Smalltalk system such as Pharo (https://pharo.org/) or Squeak (https://squeak.org/).

Rights

Copyright © 2020–2025 Gregor Richards.
This module is intended for CS442 at University of Waterloo.
Any other use requires permission from the above named copyright holder(s).