I Can Read C++ and Java But I Can't Read Smalltalk

Wilf LaLonde

Journal of Object-Oriented Programming, February 2000, pp. 40-45, http://www.joopmag.com

Introduction

I've had a number of people tell me that they know C++ or Java very well but are completely mystified by Smalltalk. According to them, Smalltalk just does not make sense! I thought about it for a while and came to the conclusion that they might just be right. If I pick a random number of columns I wrote over the years and pretend I knew just Java, I'm sure I wouldn't be able to understand the code. There are some very simple syntactic notions that need to be mastered before Smalltalk can be understood but there are also some very subtle and sophisticated notions. If "Johnny can't read Smalltalk", perhaps I can make a difference. What I'll try to do is to bring you up to speed in stages. I am assuming that you already understand object-oriented programming, however. For those of you who already know Smalltalk, pretend you don't for a while.

The Oh So Simple Lexical Details

There are some obvious little details like double quotes for comments, single quotes for strings, and special syntax for characters (e.g., $x for character "x") that might confuse you on first reading since the conventions might be different from that used by other languages. There is also the notion of a symbol which is a string that is unique memory-wide; i.e., when it is constructed (typically at compile-time), a memory search is made to determine if another one like it exists; only the original is used. The rationale is not memory saving but significant speed-up when comparing symbols (explained below).

```
"this is a comment"
'this is a string'
#'this is a symbol'
#thisIsASymbolToo
```

There are very minor distinctions between the notations used to denote assignment and comparison operators.

```
:=    //Means assignment
=     //Means equality comparison
==    //Means identity comparison
```

If you give me two objects referenced by distinct variables "a" and "b", I can tell you if they are the same object (via a == b) or distinct objects that look the same (via a = b). Intuitively, == compares two pointers whereas = compares the entire contents of the objects.

There are also very few commas evident in Smalltalk programs because they play no syntactic role. That's why array literals, for example, are comma-free; e.g.,

```
#(1 2 3 4 5)
```

However, comma is an operator. So you will see it used now and then to concatenate two strings; e.g.,

```
'string1', 'string2'
```

Keywords Are Pervasive

<PULL QUOTE from below "Keywords are everywhere in Smalltalk."**>**
Keywords are everywhere in Smalltalk. But they serve to help readability, not hinder it. To see why, let's start with the notation used in C++ and Java. For example, you might be totally at ease with an example such as

```
t->rotate (a, v);      //For C++
t.rotate (a, v);      //For Java
```

Object t is sent the message rotate with parameters a and v. To understand the code, readers would generally seek out the variable declarations to determine their types. Let's assume they were declared as follows:

```
Transformation t;
float a;
Vector v;
```

In Smalltalk, variables can refer to any type of object. So types are not needed although we do need to declare the variables; e.g.,

```
| t a v |
```

In the absence of declarations, good programmers in Smalltalk use names that indicate the type. So we would generally write the following declaration instead.

```
| aTransformation angle aVector |
```

But allow me to continue with short names just so that my examples won't wrap around in the Journal's narrow columns. Let's "evolve" the C++ and Java syntax by eliminating what's not necessary. For example, can you still make sense of the following?

```
t.rotate (a, v);  //Original
t rotate (a, v);  //Who needs dot?
t rotate a, v;     //Who needs brackets?
```

To evolve the syntax further, we need to know what the two parameters a and v represent. Let's assume the entire example is meant to "rotate by angle a around vector v." Then the next step might be

```
t rotate by a around v; //Who needs commas?
```

How do we tell what the components are? We know, because we evolved the example, that "t" is a variable, "rotate" is the name of the method, "by" is a separator, "a" is a variable, "around" is a separator, and finally "v" is a variable. To eliminate potential ambiguity, we can establish a convention: add colons after separators. Then we get

```
t rotate by: a around: v; //Who needs ambiguity?
```

Finally, let's insist that the separators be part of the method name; i.e., let's require that the name of the method be "rotate by: around:" and let's get rid of the spaces to get "rotateby:around:" as the name and finally, let's capitalize internal words just for readability to get "rotateBy:around:". Then our example could be written

```
t rotateBy: a around: v //This is Smalltalk
```

So the method name is broken up into pieces. Fortunately, it's easy to mentally assemble the pieces into one complete name. When the method is defined in a class, we might be tempted to define it as follows.

```
self rotateBy: angle around: vector
  | result |
  result := COMPUTE ANSWER.
  ^result
```

At execution time, there is a one-to-one mapping between "t" and "self", between "a" and "angle", and between "v" and "vector." Note that "^" indicates that an answer is returned; it's the Smalltalk equivalent to the "return" keyword. Variable self is a synonym for "this" and "^self" is executed by default if the end of the method is reached without an intervening return; i.e., you can't fall off the end of the method even if you forgot to return an answer. This also implies that every method returns an answer even if it's not used by the sender.

In point of fact, however, correct Smalltalk syntax requires that "self" not be present in the method header (rather, it MUST be implied) as follows:

```
rotateBy: angle around: vector
  | result |
  result := COMPUTE ANSWER.
  ^result
```

The nice thing about keyword syntax is that we can define different keywords for different methods. For example, we can define a second method used as follows:

```
t rotateAround: vector by: angle
```

There is no need to memorize the order of the parameters. The keywords tell us the order. Of course, as programmers, we can abuse keywords. For example, if we defined the method with the keywords as follows:

```
t rotate: angle and: vector
```

the reader would not be able to tell if the parameters were in the correct order. That's just poor style. If there is only one parameter, then so be it. We still need a method name that has one keyword; e.g.,

```
t rotateAroundXBy: angle
t rotateAroundYBy: angle
```

We like to think of keywords (easily spotted because they end in colon) as introducing parameters. But what do you do if there are no parameters.

```
t makeIdentity: //Can a colon at the end make sense?
```

If keywords introduce parameters and there are no parameters, then let's not use a keyword. So a zero-parameter message would really appear as

```
t makeIdentity//This is Smalltalk
```

Of course, binary operators are also allowed but there are no unary operators (makeIdentity above is a unary message but not a unary operator). When many message types are mixed together, we might end up with an expression like

```
a negative | (b between: c and: d)
   ifTrue: [a := c negated]
```

As a reader, you should be able to tell that "a" is sent message "negative" (no parameters) which comes back true or false; "b" is sent message "between: c and: d" which also returns true or false. The results are ORed together to become the receiver for the "ifTrue: [a := c negated]" message. This is an if-then control structure but it's not a special syntax. It's just the standard keyword syntax where the receiver is a boolean, the keyword is "ifTrue:", and the parameter is "[a := c negated]" (which we call a block). You'll never see "a := -c" in Smalltalk because there are no unary operators although you will see constants like "-5" where the "-" sign is actually part of the constant.

So if you see an expression like "-3.5 negated truncated factorial", you should instantly realize that there are no keywords. So "-3.5" must be sent the message "negated"; the result 3.5 is sent the message "truncated"; it's result 3 is sent "factorial" which produces the final answer 6.

Of course, there are rules about evaluation order (left-to-right), message priority (zero-parameter message are highest, then binary, and keyword), and such. That's really important for those writing the code, but you can get by without those details as a reader. Oh yes. Left-to-right evaluation really means that

```
1+2*3 is 9
```

There is no operator priority. But you'll rarely see a Smalltalk programmer write such an expression because he knows it will confuse non-Smalltalk readers. Instead, he will and should write

```
(1+2)*3
```

even though the parentheses are not needed.

Semicolon is Not the Same as Period

Most non-Smalltalk readers expect semicolon as the statement terminator but in Smalltalk, period is used for that purpose. So we don't write

```
account deposit: 100 dollars;
collection add: transformation;
```

Instead, we write

```
account deposit: 100 dollars.
collection add: transformation.
```

Oh! Did the "dollars" message confuse you? There is no magic. For this to work, there must exist a method called "dollars" in the Integer class that constructs a "Dollar" object and returns it. It's not in the base Smalltalk environment but we can add it. Base (or built-in) classes can be extended just like user defined one, if necessary.

So period is a statement terminator and it's optional on the last statement (so you could classify it as a statement separator too, if you want). But semicolon is also legitimate and it is a special separator (not a terminator). It indicates that the receiver is NOT REPEATED. So instead of writing.

```
| p |
p := Client new.
p name: 'Jack'.
p age: 32.
p address: 'Earth'.
```

we can write

```
| p |
p := Client new.
p
  name: 'Jack';
  age: 32;
  address: 'Earth'.
```

or better yet

```
| p |
p := Client new
  name: 'Jack';
  age: 32;
  address: 'Earth'.
```

Formatting here is irrelevant. We can put the whole thing on one line if we wish. Essentially semicolon indicates that the previous message was sent to modify the receiver and that the next message should be sent to the same receiver (not to the answer which is to be discarded and disregarded).

In this last example, “new” is sent to the class to obtain an instance (the answer). Then “name: 'Jack'” is sent to this instance. The first semicolon indicates that the result of “name: 'Jack'” is to be discarded and that “age: 32” should be sent to the previous receiver (which is the same instance). The second semicolon indicates that the result of “age: 32” should be discarded and “address: 'Earth'” sent to the previous receiver (still the instance). Finally, the result of “address: 'Earth'” is assigned to p. Methods that modify receiver typically return the receiver, so p is bound to the new heavily modified client object.

We could paraphrase the execution of the above assignment in a much simpler way by substituting the English words “AND ALSO” for the semicolon. So “new” is sent to class “Client” and the answer is sent “name: 'Jack'” AND ALSO “age: 32” AND ALSO “address: 'Earth'”. Repeatedly sending different messages to the same receiver is called cascading in Smalltalk. Semicolons can also exist in subexpressions as in “p := (Client new name: 'Jack'; age: 32; address: 'Earth')”—notice the parentheses.

Get/Set Methods Have The Same Names as Instance Variables

Variables like name, age, address in an instance of a class Client are all private in Smalltalk. But methods can be implemented to provide access. In C++ (Java is similar), for example, we might write accessing methods (often called get/set methods) such as the following.

```
long getAge () {return age;}
void setAge (long newAge) {age = newAge;}
```

If you have dozens of classes and you use this convention, you’ll have hundreds of messages that start with get and set. If, by chance, you decide to eliminate these repetitious prefixes by using shorter names for the methods (as follows), the C++ compiler will get upset because it can’t distinguish the variable from the method. The Java compiler, however, can tell the difference.

```
long age () {return age;}
void age (long newAge) {age = newAge;}
```

Can you distinguish the variable “age” from the message “age”? You should. When you use the message, you need parentheses as in “age () or age (22)”; when you refer to the variable, you don’t use parentheses. The equivalent methods in Smalltalk can be written as follows:

```
age ^age
age:  newAge  age := newAge
```

However, we generally use two lines as follows for readability:

```
age
   ^age
age:  newAge
   age := newAge
```

There are no parentheses in Smalltalk to help you distinguish the variable from the message, but it's still very easy to do. If you can do it, you'll be able to tell me how many variables there are in the following nonsense expression:

```
age age age: age age + age age
```

Well!! The answer is 3; the first age must be a variable and so must the fourth (after a keyword, a new subexpression must start; all expressions must start with a variable) and the seventh (after a binary operator, a new subexpression must start too). With simpler and more typical examples, it's dead easy. Consider

```
name size //name must be a variable
self name //name must be a message
```

Collections are Used Pervasively

The two most heavily used types of collections in Smalltalk are ordered collection and dictionaries. Arrays are conceptually ordered collections that can't change size.

```
| a b |
a := OrderedCollection new
   add: #red;
   add: #green;
   yourself.
b := Dictionary new
   at: #red put: #rouge;
   at: #green put: #vert;
   yourself.
```

In each of the above assignments, the variable is bound to the result of the last message executed; i.e., the result of "yourself" which is, hopefully, the newly created collection. Message "yourself" is designed to return the receiver (almost like a no-op) but "add:" and "at:put:" do not (they return their last parameter). So without "yourself", we would bind "a" to #green" and "b" to #vert.

I deliberately used cascading above in order to explain why "yourself" is used at all since it appears in isolated methods in the built-in classes.

The big advantage to Smalltalk collections is that you can put any objects whatsoever in the collections. Even the keys in dictionaries can be any type of object. Also, the objects in a specific collection don't have to be the same type; they can all be different types. We don't need to invent new kinds of collections just because we want to accumulate instances of a brand new class.

Ordered collections can be accessed like arrays; e.g., “a at: 1” indexed starting at 1. Dictionaries are accessed the same way too; e.g., “b at: #red”. But there are many applications where we don’t need to deal with the keys. For these, there are very simply looping facilities that iterate through the items; e.g.,

```
a do: [:item |
  USE item FOR SOMETHING].
b do: [:item |
  USE item FOR SOMETHING].
```

Loop variable “item” gets the elements of the collection one-by-one even if they are all different types. If necessary, we can easily find out what something is at execution time; e.g., “item isKindOf: Boat” returns true or false. There are also many special-case query messages like “item isCollection” or “item isNumber”. There are also many looping constructs that create and return new collections such as

```
c := clients select: [:client | client netWorth > 500000].
d := clients collect: [:client | client name].
```

In the first case, we get a subcollection of those clients that are pretty rich. In the second, we get a collection of names (the original collection was a collection of clients).

Sequencing Abstractions are Built Without Creating New Classes

Every now and then, a reader will find code that looks like

```
stuff value: x value: y value: z
```

where the keywords are all “value:”. This clearly looks useless and bewildering to a non-Smalltalk reader. What is happening is that the programmer has (typically) created a new abstraction facility.

Let me explain this fundamental capability that only Smalltalk supports with a challenge. Given that we have already introduced the Client class earlier, suppose that we want a simple facility for looping through an individual client’s parts; i.e., we want the loop to first give us the name, then the next time around, we want the age, and finally we want the address.

The conventional solution in C++ and Java is to create a special stream class or enumerator, say called ClientIterator, with facilities to initialize it, to ask it if it is at the end, and, if not, to ask for the next object in the iterator. Then we can have a loop that sets up the iterator and, while not at the end, gets the next object and processes it. The advantage of the iterator is that it can provide its own variables for keeping track of where it’s at in the sequencing process; i.e., it’s not necessary to extend the Client class with “temporary” instance variables needed for iterating.

An example piece of code that might make use of the intended abstraction is shown below:

```
aClient := CODE TO GET ONE.
aClient partsDo: [:object |
  object printOn: Transcript]
```

Notice that partsDo: looks like a looping construct where object is the loop variable. The first iteration, we get the name and print it on the transcript (a special workspace in the Smalltalk programming environment). Then we get the age in the second iteration and finally the address in the third. Also note that "partsDo:" is a keyword message where "aClient" is the receiver and "[:object | object printOn: Transcript]" (a block) is the parameter.

Before we go too far, let me just show you the Smalltalk solution. Then I'll explain how it works and provide more sophisticated examples. All we do is add one method to class Client as follows:

```
partsDo: aBlock
  aBlock value: self name.
  aBlock value: self age.
  aBlock value: self address.
```

The clue to understanding this is the realization that blocks are nameless functions. To better understand what I mean, suppose that we want to assign a function to a variable BUT WE DON'T WANT TO CALL IT. Let me make up the syntax for a C-like language as follows (I do know the correct syntax for doing this in C but it doesn't help to elucidate the important idea; so I'm not using it):

```
a = f (x) {return x + 1;} //C-Like syntax
a := [:x | x + 1]          //Smalltalk syntax
```

Now variable "a" is the function object itself. Since f is a function, we might expect to be able to call it with the notation "f (10)" to get back 11. But we should also be able to call it by executing "a (10)" because a's value is the function. Executing the function through variable "a" doesn't require any knowledge of its original name.

So in Smalltalk, we don't even bother with a name for the function. We simply assign it to any variable and use it through that variable. To "call" the function, we unfortunately don't have the nice notation used above. We have to settle for "a value: 10" which returns 11. During execution, parameter x is bound to 10, x + 1 is computed, and the end of the block is reached. When that happens, the last value computed is returned.

In general, we rarely execute blocks directly. Instead, we write abstractions like "partsDo:" above and hide the cumbersome "call" to the block in the method providing the abstraction.

Consider some more examples. Suppose we have an Airplane class that maintains a list of passengers. Let's just suppose that we want to sequence through those passengers that are kids (suppose we define kids to be 12 years old or younger). Then a piece of code for doing that might look like

```
anAirplane passengers do: [:person |
  person age <= 12
    ifTrue: [..DO SOMETHING with person..]]
```

If we need to sequence through kids in other contexts, it becomes useful to create an abstraction that simplifies our code. All we need to do is implement a “kidsDo:” abstraction as follows in class Airplane (I’ve added line numbers for ease of reference):

```
1.  kidsDo: aBlock
2.    "Here, self is an Airplane. "
3.    self passengers do: [:person |
4.      person age <= 12
5.        ifTrue: [aBlock value: person]]
```

Then we would modify the example to use the new abstraction as follows:

```
6   6.  anAirplane kidsDo: [:kid |
7.      ..DO SOMETHING with kid..].
8.    "All done. "
```

Can you see how the code beginning with line 6 works? When the “kidsDo: …” message at 6 executes, the “kidsDo:” method in line 1 is invoked. So variable aBlock in line 1 is bound to “[:kid | ..DO SOMETHING with kid..]” (let me call this the kid block). Inside the kidsDo: method, “do:” in line 3 iterates through all passengers. In line 5, aBlock is sent a “value:” message only when a person is found that is 12 years old or younger. When this “value:” message with “person” as parameter executes, a function call is made to the kid block which causes “kid” to be bound to “person” and “..DO SOMETHING with kid..” in line 7 to execute. When the end of the block is reached, execution goes back to the “do:” loop in “kidsDo:” (the end of line 5) as the loop continues to look for other kids. When the loop finally finished, execution returns from the “kidDo:” method invocation in line 6 and we reach line 8.

To recap, code in 6 causes code in lines 1 through 5 to execute which repeatedly causes code in the kid block (line 7) to execute.

In general, blocks provide Smalltalk with one of the simplest and cleanest facilities for building control abstractions. They are also designed to execute return statements with semantics that are surprising when initially described (I haven’t yet described it). Nevertheless, it turns out to be the only possible semantics that works. Let me describe what I mean by adding a new Airplane method containing code similar to that in lines 6-8 as follows:

```
10. findAnySickKid
11.   "Here, self is an Airplane too. "
12.   self kidsDo: [:kid |
13.           kid isSick
14.             ifTrue: [^kid]].
15.   ^nil "None exists.”
```

If you read the code, I don’t believe you’ll find anything unusual. A loop is initiated over all kids on the airplane. If a sick one is found, it is returned. Otherwise, further iterations continue. If there are no sick kids, the loop will eventually terminate and nil (a special object that can easily be tested for) is returned. So what’s interesting? Well, there are 3 important execution locations: line 10 which starts

method findAnySickKind, line 1 which starts method kidsDo:, and finally lines 13-14 which is the body of the kid block. By executing “anAirplane findAnySickKid”, I invoke method findAnySickKid which invokes kidsDo: which invokes the kid block. Inside the kid block, executing “^kid” does not return to the sender (method kidsDo) but rather to the sender of findAnySickKid. No matter how long the chain of messages is that leads from method kidsDo: to the code inside the kid block, “^kid” always returns from findAnySickKid. I call this a short circuit return although I don’t believe anyone else has given the process a name.

Conclusions

I’ve given a shot at making Smalltalk more understandable to the uninitiated. There are lots of issues that I haven’t gone into but that’s OK. If you’re new to Smalltalk, you should now have a much easier time picking up on your own. Many thanks to Alan Francis for alerting me to the fact that Java programmers might have a hard time deciphering Smalltalk, having admitted to being one himself.