Reading Smalltalk

My presentation is based on the way I tackle any language:

1. Examine the character set and tokens
2. Examine the reserved words
3. Examine each unique syntactic form
4. Examine each unique semantic form
5. Examine the libraries

So here goes...

1. Character set and tokens

Standard character set, with twelve special characters: #:^.'|";()[]

The tokens are: {identifier} {number} {string} {comment} {binaryOperator} {keyword} {specialToken}

Identifiers
are the same as you'd expect, except that we use capitalLettersLikeThis, rather_than_underscores.

Numbers
are also as you'd expect.

'Strings'
'are enclosed in single quotes'

"Comments"
"are enclosed in double quotes"

Binary operators
are composed of one or two characters. The characters which can form a {binaryOperator} vary a little bit between implementations, but for the purpose of *reading* Smalltalk, you can assume that any non-alphaNumeric character which is not in the above list of {special characters} forms a {binaryOperator}. For example,

+ is a {binaryOperator}.
++ is a {binary operator}.
?* is a {binaryOperator}.
-> is a {binaryOperator}.

A ***keyword:***
is just an identifier with a colon on the end of it, e.g. *anyIdentifierLikeThis:* is a {keyword}. In Smalltalk, a keyword is only special in the sense that it forms a "keyword message". It is a distict kind of token (different from an identifier or a string) but it's meaning as an individual token is not special. Some languages have {keywords} like BEGIN and END with builtin special meanings. {Keyword} in Smalltalk is not this sort of thing, it's strictly a syntactic form.

SpecialTokens

are just the special characters, used as seperators for parsing the language.

#	pound	begins a #symbol
:	colon	ends a keyword:
^	caret	^answerThisObject
.	period	statement separator
'	single quote	delimits a string
|	vertical stroke	delimits temp variables
"	double quote	delimits a comment
;	semicolon	statement cascade
(	openParenthesis	begins an expression
)	closeParenthesis	ends an expression
[	openSquareBracket	begins a block closure
]	closeSquareBracket	ends a block closure

2. Reserved Words

There are five reserved words: ***nil false true self super***.
These are all reserved because the compiler, optimizer, and VM know about them.

nil

is the value of any variable which hasn't yet been initialized. It is also the value of any variable whose initialization has been forgotten. It *should* be used to mean "I have no idea", "Has never had a value", or "If it ever had a value, someone has since asked that we behave as if it never had one; therefore - I have no idea". It is sometimes incorrectly used for things that should be NullObjects or ExceptionalValues .

true and ***false***

are singleton instances of the classes True and False, respectively.

self

refers to the object whose class contains the method you are presently reading, when you are reading one and encounter the word ***self***. If the object's class has no such method, you must be reading the *nearest* superclass which does have such a method.

super

refers to the same object as self.

Read that last sentence 100 times, until you accept it as fact, then move on.

So why have two names for one thing? This is a little hard to follow until you get used to it. ***super*** is the same object as ***self***, but when you try to figure out which method the object will execute in response to the message being sent, pretend the object's class didn't have such a method.

In other words, if the object's class does have a method for the message you're sending,

don't use it. Always start looking for the method in the object's superclass.

This is so you can extend your superclass' behavior without having to rewrite it.
For example, define *aMethod* that does the same thing as the superclass, and then some:

```
>>aMethod
    super aMethod.
    self doSomeMoreStuff.
```

Or, define *aMethod* to do some new stuff, and follow it up with whatever the superclass does:

```
>>aMethod
    self doSomeStuff.
    super aMethod.
```

3. Syntactic Forms

There is one overriding, but previously unfamiliar pair of concepts at work in Smalltalk:

everything is an object
and
all code takes the single conceptual form: ***anObject*** *withSomeMessageSentToIt*.

If you want to continue working in C++, Java, etc. then make very certain you do not understand what this means.
If it starts to make sense to you then by all means stop reading Smalltalk, you are in serious danger. More on this later...

There are 6 syntactic forms:

1. **Unary message send**

```
object isSentThisUnaryMessage.
```

2. **Binary message send**

```
object {isSentThisBinaryOperator} withThisObjectAsOperand.
```

3. **Keyword message send**

```
object isSentThisKeywordMessage: withThisObjectAsParameter.
object isSent: thisObject and: thisOtherObject.
object is: sent this: message with: 4 parameters: ok.
object is: sent this message: with parameters: (1 + 2).
object is: (sent this) message: (with) parameters: (3).
```

These are a little bit weirder, until you catch on. Keyword messages written as C function calls would look like this:

```
isSentThisKeywordMessage(object,andParameter);
isSentAnd(object,thisObject,thisOtherObject);
isThisWithParameters(object,sent,message,4,ok);
isMessageParameters(object,this(sent),with,(1+2));
isMessageParameters(object,(this(sent)),(with),(3));
```

Which is sort of why we *refer* to keyword messages, descriptively, like this:

```
isSentThisKeywordMessage:
isSent:and:
is:this:with:parameters:
is:message:parameters:
```

even though we actually write them as shown earlier. Note that a parameter, or the operand of a binary message, can be either an object, or the result of sending a message to an object. Just as in C, where a parameter, or the operand of an operator, can be either {an object} a literal, a constant, a variable, a pointer {or the result of...} an expression, or a function call.

4. **A block** (a.k.a. closure, or block closure)

```
[thisObject willGetThisUnaryMessageSentToIt]
[:someObject| someObject willGetThisMessage]
[:first :second| thisObject gets: first and: second]
[:first :second| first gets: thisObject and: second]
```

A block can be thought of as the only instance of an impromptu class with no superclass and exactly one method. {Not actually true, but think of it this way until you really need to understand otherwise}. What is the *one method*? Depends on the number of parameters:

If a block has		**then it's only known method is**
no parameters	["a parameterless block"]	*value*
one parameter	[:x| "a one parameter block"]	*value: actualParameter*
two parameters	[:x :y| "a two parameter block"]	*value: firstActual value: secondActual*
and so on.		

Examples:

```
[ object messageSent ] value.
```

When this block receives the unary *value* message, the unary message *messageSent* will be sent to the object ***object***.

```
[ some code ] value.
```

The *value* message causes the block to "execute" some code.

```
[ :one| any code can be in here ] value: object.
```

The *value: object* message causes the formal parameter ***one*** to be bound with the actual parameter ***object***, and the code then "executes".

5. **Answer** (a.k.a. return a value)

   ```
   ^resultingObject
   ```

   Every method contains at least one of these, even if you can't see it. Usually you can see it, and it is the last line of the method. If you can't see it, pretend you saw ^self as the last line of the method.

   The other use for this thing is the "early out", as in

   ```
       object isNil ifTrue: [^thisObject].
       object getsThisMessage.
       ^self
   ```

   This may strike you as an unusual form, as it violates the "single entry/single exit" maxim from structured programming. Keeping in mind that Smalltalk methods are typically short, no, make that *very* short, we simply don't care. The forces have changed - it's hard to get lost reading a method of just a few lines, and if later we need to make a change that affects all the exit points, well, big deal.

6. **Method definition**

   When using a browser, you don't actually see this syntactic form, but when Smalltalk is being described outside it's own environment, the following syntax is used to indicate the definition of a method:

   - **Unary**

     ```
         ClassName>>methodSelector
             someObject getsThisMessage.
             someOtherObject getsThisOtherMessage.
             ^answerYetAnotherObject
     ```

     This means that the class named "ClassName" has a method definition for the unary message *methodSelector* and it's definition is as shown.

   - **Binary**

     ```
         ClassName>>+ operand
             instanceVariable := instanceVariable + operand.
             ^self
     ```

     This means that the class named "ClassName" has a method definition for the binary message + *operand* and it's definition is as shown.

   - **Keyword**

     ```
         ClassName>>keyword: object message: text
             Transcript nextPut: object; nextPut: ' '; nextPutAll: text; cr.
     ```

     This means that the class named "ClassName" has a method definition for the 2 parameter keyword message *keyword:message:* and it's definition is as shown.

7. **Assignment**

   Ok, I lied - there are seven syntactic forms.

   In that last binary message example, you see what appears to be an assignment statement. *It is*. And it's special, for two reasons:

   1. Because it might also appear to be a binary message. But it isn't.
      And
   2. Because it doesn't follow the otherwise consistant form:
      ```
          someObject isSentSomeMessage
      ```

8. **Cascade**

   Ok, I lied again, twice. There are eight syntactic forms, and another exception to the so called "consistant form". In that last keyword message example you also see some semi-colons. The semi-colon is shorthand for

   > send *this next message* to the same ***object*** (the one that received *the last message*).

   Hence, the line from the example above
   ```
       Transcript nextPut: object; nextPut: ' '; nextPutAll: text; cr.
   ```
   means
   > send the *nextPut:* keyword message (and parameter ***object***) to the object named "Transcript",
   > then send another *nextPut:* message (and parameter *' '*) to the same object (i.e. Transcript),
   >
   > then send a *nextPutAll:* message (and parameter ***text***) to that same object,
   > then send the *cr* message to it.
   > Finally, return yourself as the result of this method. (The implied ^***self*** at the end of the method).

Operator Precedence

Everybody's favorite memorization excercise. How many combinations of precedence and associativity do you know? How many are you *supposed* to know? Here are the rules for Smalltalk:

message	**priority**
unary	highest
binary	
keyword	
assignment	lowest
otherwise	strictly left to right

And yes, you can override this with parentheses, as usual. That's it!

> Ok hold it. You're not getting away with just this.
> It doesn't even work. E.g. 3 + 4 * 5 would be 35!?!

Ah, but we do, and it does, and you're right.

> That's just goofy!

Yes, it is. Drives you crazy, for about a week. And then it's just gone, as in ***not an issue***.

That's it.

Let me repeat - That's it! That's the entire language. The only thing left is to learn the library, and the tricks and idioms of the language.

Now the astute reader is probably thinking something like

> Wait a minute. What happened to unique semantic forms? You didn't cover control-flow. And you didn't cover variables, types, allocation and deallocation, pointers, templates, virtual member functions, static methods, etc. etc. etc.

Well, such a reader would be wrong. I covered all of that. Ok, ok, you win. I never said anything about variables. That's because they have no syntactic form, other than assignment:

> instVar1 := 'aString'.

and the notation for temporaries:

> | aTemp anotherTemp |

You define instanceVariables by typing their names into a special place in a browser window, and classVariables into a different special place. There is no syntactic form that goes with it, as it's not part of the "code". There are no types, and no 'builtin' syntactic specialties like arithmetic, casting, dereferencing, etc. There is allocation, but it is always a message send:

```
SomeClassName new
```
or
```
SomeClassName aMethodWhichJustHappensToBeAConstructor
```

and there is no deallocation. When the last reference to an object ceases to exist, the object is garbage collected. You couldn't cause a *(VOID *)(0) if you wanted to. None of the rest of that stuff exists either.

> False, you say. You didn't go over the special syntax for control flow.

Yes I did. There isn't any. Turns out you don't need such a concept as control flow littering up your syntax.

> Oh don't be rediculous, of course you do. It's completely special.

Sorry to disappoint you. Remember when I said "think of blocks as if they only have one method"?

Here's where the truth comes out. Blocks also respond to a few other messages, like:

```
    [ ] whileTrue: [ ]
```

Which means "send a *message* to an ***object***". Literally send the keyword message *whileTrue:* (with it's ***parameter*** (the second block)) to an ***object*** (the first block). What do you suppose the first block does when it gets such a message? The block evaluates itself (sends itself the *value* message). If the result is true, it sends a *value* message to the second block, and then starts over. Otherwise, it just quits, and answers ***false***.

Of course Booleans also have methods for similar looking messages:

```
False>>ifTrue: aBlock
    ^nil
False>>ifFalse: aBlock
    ^aBlock value
```

False is a class, which has methods for these two messages. Since every object which is an instance of class False is by definition logically false, there is nothing to test. It effectively ignores requests to do something *ifTrue:* and always does the thing when asked to do something *ifFalse:*. Another class, True, has the same methods with the outcome reversed. (Don't think about this one too much, it will hurt you. You'll start to think Smalltalk might not be as slow as some think it is. For instance, it's faster than Java...)

Checkout the library to see how variations on this simple theme build up every control structure you've ever thought of. Except one. Nobody ever put a SWITCH/CASE semantic form into the library. Drives beginners nuts. Later you discover that your methods are always too short to care about such a thing, and when they seem to want for one, it means your design is not taking advantage of polymorphism the way it should. So you fix that instead...

One last piece of syntactic sugar to deal with:

```
    'ThisIsAString'
    #ThisIsASymbol
```

These behave pretty much the same, except that the latter is guaranteed to be a Singleton, with a unique hash value. Useful for table lookups and such, but otherwise you can ignore it.

Hope this helps in your attempts to read Smalltalk. But be careful! The minute you get an inkling of what this all means, you'll find it very difficult to continue to use whatever language you're using now... Bar none. You've been warned ;-)

What next?

To explore Smalltalk further, you should:

Install

Dolphin Smalltalk from Object Arts.
Absolutely brilliant piece of work. Play with it, try the examples, read the education center material.
(It's all free - until you're addicted)

Read

comp.lang.smalltalk
Smalltalk: Best Practice Patterns (Kent Beck)
Smalltalk Companion to Design Patterns (Brown, et al)
AND any other Smalltalk books at your favorite bookstore.

Master

VisualWorks from Cincom Systems.
Absolutely incredible toolset, the flagship of the Smalltalk industry, power and stability which defies belief.
An *immense* class library (a.k.a. work already done for you).
(The non-commercial version of VisualWorks is also free).