Chapter

6

Special Variables, Characters, and Symbols

Smalltalk is a very small language, with a very large class library. But even though it is a small language, there are still a few things to learn.

Character Pairs

There are several character pairs that mean things when used together. The VisualWorks Browser makes it easy to manipulate text and character pairs. You can surround a block of text with a character pair by highlighting the text then pressing *Esc-leftCharacter* (the Escape key followed by the left character of the pair). You can select and highlight all the text between a character pair by double-clicking between the character and the text (at either end). You can remove the character pair by highlighting the text between the pair then pressing *Esc-leftCharacter*.

"This is a comment"

Text between pairs of double quotes is comment text and will not be compiled. You can make a block of text into a comment by highlightin

[some code]

Surrounding code with square brackets creates an instance of BlockClosure. See the section below on Automatically Constructed Objects.

Prefixes

`$` The dollar sign is a prefix for creating an instance of class Character. See the section below on Automatically Constructed Objects.

`#` The pound/hash/number symbol is used to create a symbol or a literal array. See the section below on Automatically Constructed Objects. The # character is also used in documentation to denote a message. For example, if you read an article that describes sending the `new` message, it may refer to the message as `#new`. In this book we are denoting messages with `(this font)` rather than with a # prefix.

The pound/hash/number symbol is used to create a symbol or a literal array. See the section below on Automatically Constructed Objects. The # character is also used in documentation to denote a message. For example, if you read an article that describes sending the `new` message, it may refer to the message as `#new`. In this book we are denoting messages with bolding rather than with a # prefix.

Special Variables

The special variable *self* is the receiver of the message, or the object executing the method. You use *self* when an object wants to send a message to itself. This is described more fully in Chapter 2, Messages.

The special variable *super* refers to the superclass of the class that *defines* the currently executing method. This is described more fully in Chapter 2, Messages.

The special variable *thisContext* is the current context that is executing. The statement `thisContext receiver` gives the object executing the method, while `this Context sender` gives the context of the method that sent the message being executed, and `thisContext sender receiver` gives the object that sent the currrently executing message.

The most common usage of *thisContext* is in the debugger since it is possible to trace through the stack using the `sender` message. To get a feel for *thisContext*, when you are comfortable writing methods, put the following in a method:

```
Transcript cr; show: thisContext printString.
```

Assignment

`:=` is used to assign a value to a variable. For example,

```
number := 3.
```

Assignments are the last thing to happen in a statement (except for returns). So, the following assigns the string '68' to the variable x.

```
x := 67.8 rounded negated printString copyFrom: 2 to: 3.
```

The assignment operator is usually called 'gets' or 'colon equals' when speaking. Giving it a special name rather than 'equals' means that you are less likely to type something like `number = 3`, which sends the = message to number with 3 as the parameter, and returns a Boolean.

Note that you can do multiple assignments in a statement, although I don't particularly recommend it since it doesn't help clarity. For example,

```
x := y := 3.
```

Return

The caret symbol (^) is the return symbol. If the line being executed says `^3`, the method will exit and return a SmallInteger with the value 3. ^ is the very last thing done in the statement. In the following example, the variable *instVar* will be assigned the value -3, then the value of instVar (ie, -3) will be returned from the method.

```
^instVar := 3.14 truncated negated
```

By default, a method returns *self*. That is, if the method gets to the end without executing an explicit ^ statement, it returns *self*. You can see this by holding down the Shift key when you select the method with the mouse. This shows the decompiled code, which will usually have `^self` at the end.

You will sometimes see fairly complex looking return values. Two common return statements follow. In the first example, you might think of returning from the blocks or setting a temporary variable, but the example shows the more common style. In the second example, we return the results of building a new collection in one operation rather than putting the new collection in a temporary variable then returning the temporary.

```
^objectOne > objectTwo
   ifTrue: [objectOne]
   ifFalse: [objectTwo]

^someCollection
   collect: [ :each | self manipulate: each]
```

Statement Separator

A period (`.`) is the statement separator. If there are two statements, the first needs a period to let the compiler know that the first statement is done and the second statement is about to start. Leaving out periods is one of the most common errors and usually results in a messageNotUnderstood exception when Smalltalk treats the first word of the second statement as a message to be sent to the result of the first statement. A good way to check for missing statement separators is to run the formatter after accepting a method. If the method looks different than you expected, you may have a missing period.

Note that the period is *not* a statement terminator. A method with one statement does not need a period. Nor is a period needed for the last statement in a method or a block. This is unlike C++ where a semicolon is a statement terminator and is required for all statements. Putting in a period after a statement where it is not needed is okay though. My usual mode of typing is to not worry about extra periods. After I'm done and the method compiles (ie, it is successfully accepted), I run the formatter, which removes any unnecessary parentheses and periods.

Cascading

The cascade operator is a semicolon (`;`). Cascades allow you to send multiple messages to the same object (the same receiver) without having to name it every time. The most common occurences are when printing to the Transcript, when overriding `printOn:`, and when adding multiple items to collections. For example,

```
Transcript cr; show: 'this is a string'.

aCollection
   add: 'this';
   add: 'that'.

printOn: aStream
   aStream
      print: self class;
      crtab;
      print: self instVarOne.
```

If you use cascades, the generally accepted practice is to put the message receiver on a line by itself, then each cascade goes on a separate line, tabbed over one from the receiver. Short messages such as `cr`, or `tab`, are often put on the same line in short cascades. Unfortunately the formatter destroys any indentation you specify for cascades. Because of this, I will often specify the message receiver each time and use a period to separate the statements. There is no performance loss by doing this.

Automatically Constructed Objects

There are some objects that are created automatically rather than by explicitly sending a message such as `new` to a class.

SmallInteger

`4` An instance of SmallInteger can be created by simply using the integer value. One of the interesting thing about integers is that you can't get integer overflow because if you add one to the largest possible SmallInteger, the return from the + message is a LargePositiveInteger. Try inspecting `SmallInteger maxVal + 1`.

Float

`3.14` An Instance of Float can be created by using a floating point value.

String

`'here is a string'` An instance of String (actually a subclass of String) can be created by enclosing a sequence of characters in single quotes. String is a funny class because even if you do `String new` you still end up with an instance of a subclass of String. To create a string with an embedded quote, use two single quotes to denote the embedded quote. For example,

```
'You can''t do that'.
```

Character

`$A` An instance of Character can be created by preceding the character with a dollar sign. For example, the character `X` is created by writing `$X`. You can create instances of space, tab, carriage return and other white-space characters the same way; for example, you can refer to a space as `$` and a newline by typing `$` followed by pressing the return key. However, it's difficult to read white space characters created like this, so the preferred way to create them is to send the appropriate message to Character (eg, `Character space` or `Character cr`). To see which characters can be created with message sends, look at the class side messages of Character or inspect `Character constantNames`.

Symbol

`#notFound` An instance of Symbol can be created by prefixing a sequence of characters with a #. If you want the symbol to contain space characters, you can enclose the symbol name with single quotes. For example, #'not found'. If you inspect this symbol, it will display with the quotes but when you look at the first character, you will see that it is a $n rather than a single quote.

Array

`#(1.1 $a 'hi')` An instance of Array can be created by enclosing other automatically constructed objects between parentheses and preceding this with a # character. Note the space between array elements. Note also that the array elements do not have to be of the same type.

You cannot use this type of contruction with objects that require message sends. So you *can't* say #(1 (Character cr)) and expect to get an array of an Integer and a Character. However, you *can* create literal arrays that contain other literal arrays, such as #(1 $a #(1.1 'hi' #(2 #symbol))).

BlockClosure

`['Hi' echo]` An instance of BlockClosure can be created by enclosing code between square brackets. The code will not be executed until the block is send a message from the `value` family. You can set up code in a BlockClosure then pass the block to another method where it will be executed. BlockClosures are often stored in Dictionaries for later access and execution. Note that the value of a block is the value of the last statement evaluated in the block. So, the value of [3. 4. 5. 1] is 1.

Blocks can have parameters, which are defined with a colon in front of them and a vertical bar separating them from the rest of the block. For example [:parameter | parameter echo]. Blocks can also have temporary variables, which are enclosed between vertical bars. For example, [:parameter | | temp | temp := parameter * 2. temp echo].

If all the blocks used as either receivers or parameters in messages such `whileTrue:`, `ifTrue:`, and `and:` are literal blocks (i.e., defined directly with square brackets and not stored in a variable), the blocks will be compiled in-line and are not treated as blocks.

There are three kinds of BlockClosures: full, copying, and clean. In general, blocks need to maintain a reference to the method that defined them, which they store in a variable called *outerScope*. If the block contains an explicit return (^), or a reference to variables defined outside the block and which can change after the block

is created, outerScope will contain a non-nil value and the block is a *full* block. If there is no explicit return and any outside variables are determined to be non-changing after the block is created, copies of these variables are made and stored in another instance variable called *copiedValues*. For obvious reasons, this type of block is called a *copying* block. A block that contains neither an explicit return or references to outside variables is called a *clean* block. So, for example:

```
[ :parameter | parameter echo ]              clean
[ :parameter | | temp | temp := parameter ]  clean
[ :parameter | self foo: parameter ]         copying (note the self)
[ methodTemporary echo ]                     copying or full
[ :parameter | ^ parameter * 3 ]             full
```

Because full blocks have to chain back through the outer scope pointers, and this chaining is relatively expensive, it pays to avoid full blocks if possible. Create clean blocks if you can since they have the highest performance. If your block refers to *self* or to another outside variable, try to make it a copying block — ie, convince the compiler that the variables will not change after the block is created.

= and ==

The two messages = and == are not special in the same way as the things described above; they are just binary messages like any other binary message. However, there are some things about them that deserve to be specifically mentioned, so here is as good a place as any.

The = message returns *true* if the two objects are equal, while the == message returns *true* if the two objects are *identically* equal. What does this mean? (The opposites of = and == are ~ and ~~. These are a single tilde and two tildes.)

Identically equal

Objects are *identically equal* when they are the *exact same* object. The way that Smalltalk knows they are the same object is that they have the same object pointer. Object pointers are 32 bit quantities, with some of the bits providing information about whether the value is a pointer or an object value. If you have an employee object, the object pointer will reference the employee object that exists somewhere in memory. Two employee objects are identically equal if the object pointer references the same location in memory.

On the other hand, some objects can be represented fully in 32 bits. These objects return *true* when sent this `isImmediate` message, and currently are objects of class *SmallInteger* and *Character*. (The maximum value of SmallInteger is less than the maximum value of a 32 bit integer because three bits are used to denote that it is a SmallInteger and not a pointer to an object.) Because of the way they are stored, identically named instances of Symbol are also identically equal.

```
3 == 3                  true
$a == $a                true
#abc == #abc            true
true == true            true
nil == nil              true
'abc' == 'abc'          false
3.14 == 3.14            false
#(1 $a) == #(1 $a)      false
```

If you examine the following, you will discover that *x* is identically equal to *y*. This is because assignment does not copy the value; it binds the same object to a new variable so that both variables now contain the same object.

```
x := #(1 $a).
y := x.
x == y                    true
```

Equal

Objects are *equal* when their values are equal. Two strings with the same character sequence will be equal. If we look at the examples above that returned false and check for simply equality, we get

```
'abc' = 'abc'             true
3.14 = 3.14               true
#(1 $a) = #(1 $a)         true
```

If you define a new class, how does the default = method tell whether you consider two object of this class to be equal? The answer is that it doesn't. If you look at the default = method defined by Object you will see that the default implementation requires two objects to be identically equal before it agrees they are equal. Ie, = simply does `^self == anObject`.

If you define your own = method you should also write a `hash` method, and vice versa. Two equal objects should also have the same hash value. The reason that `hash` and = need to go hand in hand is that some collections store objects using a hashing scheme (such as Dictionary, Set, and Bag). Because hashing schemes can have location conflicts, a hash scheme uses `hash` to find the primary location for the object, then uses = to see if an object in the location is the object it wants. (By the way, don't try to redefine ==. First, the system will ignore your override, and second, == has a well-defined meaning that should not be changed.)

If you override =, you might consider first checking that the class of the receiver is the same as the class of the parameter. The first example below checks that the classes are the same, while the second allows the parameter to belong to a subclass of the receiver.

```
MyClass>> =
   anObject class == self class ifFalse: [^false].
   ....

MyClass>> =
   (anObject isKindOf: self class) ifFalse: [^false].
   ....
```