Chapter

2

Messages

As we said in the previous chapter, we send *messages* to objects to get them to do things. There are three flavors of message in Smalltalk.

Unary Messages

A *unary* message is a message with no arguments. For example, to negate the sign of an integer we would send the unary message, `negated`.

```
4 negated
```

Binary Messages

A *binary* message looks just like an arithmetic operator. It gets the name *binary* because there are exactly two objects involved.

```
5 + 3
```

Here we are sending the + message to 5 with 3 as the argument. Unlike most languages where +, -, *, etc., are operator symbols associated with algebraic operations, in Smalltalk they are messages with exactly one argument. The binary messages are

```
+  -  *  /  **  //  \\  <  <=  >  >=  =  ~=  ==  ~~  &  |  ,  @  ->
```

Many of these are obvious, so I'll just explain the not so obvious ones.

- `**` Exponentiation
- `//` Integer division, rounding to the next lowest number.
- `\\` Modulo, returning the remainder after the division.
- `~=` Not equal.
- `==` Identically equal — ie, the same object.
- `~~` Not the same object.

`&` Logical AND, returning true if both the receiver and the argument are true. The receiver and argument are both evaluated, so it may be better to use `and:`, which only evaluates as much as is necessary and is also compiled in-line. Note that for both `&` and `and:`, if the receiver is true, the argument is returned, which may or may not be a Boolean.

`|` Logical OR, returning true if either the receiver or the argument are true. The receiver and argument are both evaluated, so it may be better to use `or:`, which only evaluates as much as is necessary and is also compiled in-line. Note that for both `|` and `or:`, if the receiver is false, the argument is returned, which may or may not be a Boolean.

`,` Concatenate two collections. Usually used to concatenate strings.

`@` Used to create an instance of the Point class.

`->` Used to create an instance of the Association class.

Keyword Messages

We still need to be able to other types of messages with one argument and messages with more than one argument. The *keyword* message lets us do this. For example:

```
'elephant' copyFrom: 3 to: 5
```

gives the string 'eph' (Smalltalk collections are 1 based rather than 0 based as in C and C++; ie, the first element in the collection is referenced by index 1). The colons separate out the keywords in the message, where each keyword takes an argument. (In strict terms, `copyFrom: 3 to: 5` is the message and `copyFrom:to:` is the message selector, but we will also refer to `copyFrom:to:` as the message.)

Message chaining

Methods always return an object (more on this later). This means that you can *chain* messages together, because there is guaranteed to be an object to send each message to. For example, the following returns -3.

```
3.14 truncated negated
```

When the floating point number receives the `truncated` message, it returns a SmallInteger, which in turn returns another SmallInteger when sent the `negated` message. Another example might be a string that contains a number. We want to change the sign on the number and convert it back to a string. One option would be to say:

```
number := '42' asNumber.
negatedNumber := number negated.
string := negatedNumber printString.
```

However, because each method returns an object, we can write this as:

```
string := ( ( '42' asNumber ) negated ) printString.
```

or we can leave out the parentheses since we are dealing only with unary messages, all of which have the same precedence.

```
string := '42' asNumber negated printString.
```

Message precedence

Unlike C++, which has very complicated rules of precedence, Smalltalk has very easy rules:

1. Evaluation is done left to right.
2. Unary messages have the highest precedence.
3. Binary messages have the next precedence.
4. Keyword message have the lowest precedence.
5. You can alter precedence by using parentheses.

One thing that is immediately different from most languages is that there is no algebra. + and * are not algebraic symbols — they are simply messages. Using the precedence rules above,

```
1 + 2 * 3
```

equals 9, not 7. To get the result you would expect, you have to use parentheses to specify the precedence.

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

To give two more examples, `2 + '4' asNumber max: 5` gives 6, because by the above rules, the unary `asNumber` is sent first, giving `2 + 4 max: 5`. Then the binary + is sent, giving `6 max: 5`, which returns 6.

The computation `30 max: 3 + 4 * 5` gives 35. There are no unary messages, so the binary messages are sent in left to right order. With the first message sent, we get `30 max: 7 * 5`. With the next binary message send, we get `30 max: 35`, which returns 35. To get an answer of 30, as you would expect from normal algebra, you would have to use parentheses such as `30 max: 3 + (4 * 5)`.

What happens when a message is sent

Whan a message is sent to an object, the Smalltalk system looks to see if a method with that name exists for that type of object (in other words, has been written and stored in the class of the object). If there is a method, it is executed. If no method of that name is defined in the object's class, the system looks in the method dictionary for its immediate superclass. If there is no method with that name in the superclass it looks in the superclass's superclass. Figure 2-1 illustrates this.

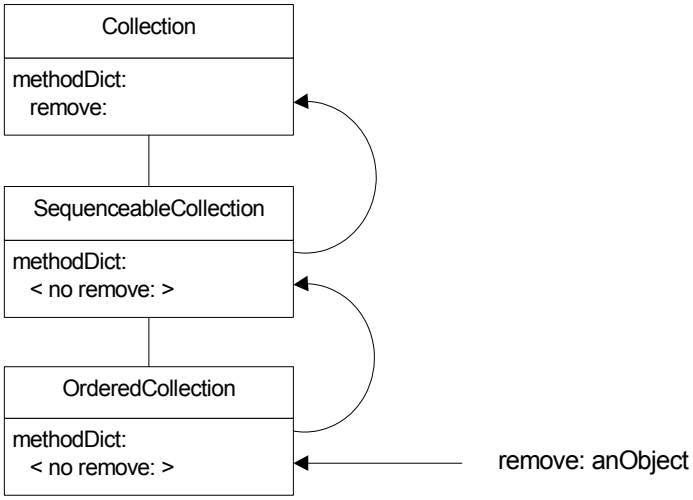

Figure 2-1.
The method lookup mechanism.

The method lookup keeps working its way up the superclass hierarchy until it finds a method with that name, which it then executes. If it reaches Object and still doesn't find a method, it displays a Notifier window that gives you an opportunity go into a Debugger and figure out what went wrong[1].

The receiver of the message

All messages have to be sent to an object — there's no such thing as a message in isolation. If you create an object in a method, it's easy to send a message to the object. For example:

```
MyClass>>doThis
    array := Array new: 3.
    array at: 1 put: 2.
```

(The generally used notation to show instance side method names is *ClassName>>methodName*. For class side methods, the notation is *ClassName class>>methodName*.)

self

Smalltalk methods average about seven lines, so for an object to do any serious work there's a good chance that you will have split the work into several methods (assuming you want to have short methods). How does a method invoke another method defined on the same object? Answer: the object sends a message to itself. Smalltalk has a special variable for just such use — *self* — which always refers to the object that is executing the code — the message receiver. Note that *self* refers to the receiver even if the the code was defined on a superclass of the receiver's class.

```
MyClass>>processObject: anObject
    self doThisWithObject: anObject.
    self doThatToObject: anObject.
```

[1] What actually happens is that if a method is not found, the `doesNotUnderstand:` message is sent. Unless this has been overridden, the `doesNotUnderstand:` method tells `Object messageNotUnderstoodSignal` to raise an exception. Unless you have a `handle:do:` block to trap this exception, the exception is unable to find a signal handler and it gets converted into another exception (an unhandled exception exception) which invokes the EmergencyHandler. Unless you have overridden the behavior of the EmergencyHandler, it opens a Notifier window.

If a method needs to call on a support method to do some work, send the message to *self*. In fact, a good rule of thumb is, if you can't figure out what object to send the message to, send it to *self*.

super

If you remember how message lookup works, Smalltalk looks for the method first in the object that is receiving the message. If it can't find a method there, it next looks in the superclass, etc. But what do we do if we explicitly want to start looking in our superclass? Smalltalk provides another special variable, *super*. So, if you want to start at your superclass, send a message to *super*.

When would this be useful? One common example is during instance creation. If you want to initialize some instance variables you usually write an `initialize` method on the instance side. You can no longer inherit `new` since it doesn't send `initialize`, so you have to write your own `new` method, which will inherit the behavior of `new` from a superclass. Note that the caret (^) shown below means *return*.

```
MyClass>>initialize
    ... set some variables ...

MyClass class>>new
    ^super new initialize
```

In fact, *super* does *not* refer the the superclass of the object that received the message. Instead, it refers to the superclass of the object that defined the code being executed. It's a subtle difference but an important one because if it were not this way it would be easy to end up in infinite recursion. Let's look at why. Let's say we have a class hierarchy with ClassTwo subclassed off ClassOne, and ClassThree subclassed off ClassTwo as shown in Figure 2-2.

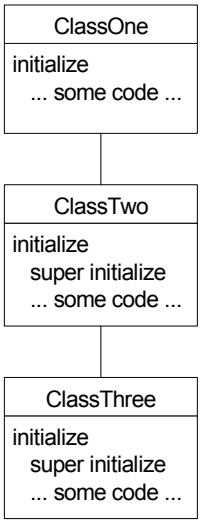

Figure 2-2.
The ClassOne, ClassTwo, ClassThree inheritance hierarchy.

All three classes have instance variables that must be initialized and the initialization code looks like the following.

```
ClassOne>>initialize
    ... set some variables ...

ClassTwo>>initialize
```

```
        super initialize.
        ... set some variables ...

    ClassThree>>initialize
        super initialize.
        ... set some variables ...
```

When we create an instance of ClassThree and execute the ClassTwo initialize code from the ClassThree object, what does *super* refer to? If it refers to the superclass of the class *executing* the code, then *super* will be ClassTwo and the `initialize` message will be again sent to ClassTwo. Ie, we'll end up in an infinite loop. On the other hand, if *super* refers to the superclass of the class *defining* the code, the message will be sent to ClassOne and everything works fine.

A key point to note is that *self* has an identity of its own and can be inspected, assigned to a variable, and passed as a parameter. However, *super* has no identity and you cannot inspect it or assign it. When you accept a method, the compiler compiles the method text into byte codes. When it comes across the word *super* it generates byte codes that instruct the run-time engine to start the method lookup in the superclass of the class defining the method. Thus *super* is simply a mechanism for telling the compiler how to generate byte codes.