Chapter

3

Methods

Methods contain code that is executed in response to message sends. Unlike functions in C or C++ which can be void functions, in Smalltalk all methods return a value. If there is no explicit return value, the method will return `self`, the object executing the method.

Method Types

There are two basic type of method: ones that return an object other than `self`, and ones that cause an effect. Examples of the first type are `asSortedCollection`, which converts a collection into a sorted collection and returns this, and `printString`, which returns an object that is a printable representation of the object receiving `printString`. Examples of the second type of method are printing to the Transcript (eg, `Transcript show: 'hello'`), and updating a file with data.

As a general philosophy, it's better to try and make your methods one type or the other. Try to avoid methods that both do something and return a meaningful object. Sometimes it will be unavoidable, but less often than you might expect. Unlike languages where you have to check the status returned from a function call, Smalltalk provides an exception handling mechanism that allows you to bypass the whole status checking aspect of error handling.

Method Length

In the VisualWorks class library, the average Smalltalk method length is reputed to be seven lines. A short method is a good method because it makes it easy for programmers to understand what the method is doing. A short method is also easier to reuse and easier for subclasses to override in order to modify the behavior. For these reasons it is a good idea to define a "standard" sized browser and have a guideline that all methods should fit in the browser window. If a method is larger, it's usually possible to break it into smaller methods by abstracting out the concepts of what you are doing and putting each concept in its own method. For example, a long method may end up as a short method such as:

```
MyClass>>myMethod
```

```
self myDoConceptOne.
(someCondition)
   ifTrue:
      [self myDoConceptTwo]
   ifFalse:
      [self myDoConceptThree.
      self myDoConceptFour]
```

Method names

Because short methods that fit in a screen can be more easily understood, it doesn't make sense to prevent that understanding with poor names. Use method and variable names that are as explicit as possible. Don't abbreviate words unless they are standard abbreviations understood by everyone working on the project. Since you don't type names very often in Smalltalk, it doesn't hurt to make them long enough to be very explicit and understandable.

Method names should say exactly what the method does. Don't be stingy with the number of characters; it's more important to be understandable than short. A programmer should be able to pick up someone else's code and immediately know what the method does, just be reading the name. However, it's not always easy to come up with a great method name when you are looking at the method itself. It's when the method is being invoked, when you are doing the message send, that you'll get the best idea of how good the name is. If it makes sense in the context where it is sent, then you've probably got a good name. So, while it's okay to name the method when you are writing it, be prepared to go back and change the name after you use it in other methods.

Stupid methods

If a method is short, has a good name, and has well thought out names for its variables, it will be fairly understandable. If class comments describe the general behavior of the object and describe the instance variables well, a good method doesn't need much comment. I really like the idea of *stupid methods*. (I'd like to give credit to the creator of the term but unfortunately I can't remember where I read about the idea.)

A stupid method is one that is so obvious that it doesn't need any documentation. The problem with documentation is that it makes a method longer, so if you have a method that is long enough to need some serious documentation, you compound the problem by making it longer and less able to fit on a screen. Not fitting on a screen, it is less readily grasped by a programmer. The ultimate compliment of a method is "that method is really stupid."

Method formatting

The formatter does a pretty good job of formatting methods for you. My rule is to ignore formatting when I'm creating a new method, then to run the formatter. If I don't like what the formatter does, I'll reformat the code by hand. Here are a few examples of how I might reformat when I don't like the formatter results. To save space I won't show the original formatting; to see it, you'll have to type in the code and run the formatter.

```
myMethod: aParameter
   aParameter
      doThis;
      doThat;
      doSomethingElse
```

```
myMethod: aParameter
   self errorSignal
      handle: [ :ex | ex return]
      do:
         [aParameter doThis.
         aParameter doThat]

myMethod: aSource and: aDestination
   (aSource location = aDestination location)
      ifTrue: [self errorSignal
         raiseWith: #sourceEqualsDest
         errorString: 'This is an error string'].
```

Some people find the formatter's format too difficult to read, especially in the case of complex, nested code. An approach preferred by some is to put each closing square bracket on its own line. The trade-off is that it becomes less likely that a method will fit into one screen, which is a strong goal of mine. One answer is that if the code is difficult to read, the method is very likely too long, too complex, or both. So you might call the formatter a benefit if it forces you to refactor your code to make it more elegant and comprehensible.

To improve clarity it's often worth putting in redundant parentheses. A programmer looking at code and working through the message precedence rules can always figure out what is happening, but you can shorten the process if you add parentheses to group things. Since the formatter removes redundant parentheses but leaves required parentheses, you can also use it to tell you where parentheses should be. For example, the first conditional below causes an exception, the second conditional shows what you intend, and the third conditional is what the formatter gives you when you format the second conditional. If you had something more complex, it might be worth leaving the extra parentheses in.

```
a < b |  c < d
(a < b) |  (c < d)
a < b | (c < d)
```

Public and Private methods

In C++ there is the notion of public and private fuctions. Public functions are ones that any object can invoke. Private functions are ones that are accessible only to instances of the class that defines them, or its subclasses (actually protected functions in the latter case).

Smalltalk does not have such a concept. All methods are accessible to all objects — they only have to send the correct message. However, the notion of public and private methods is useful; you really don't want other objects sending messages that exist only to help with the implementation details. Suppose you decide to change the way something is implemented. You don't want to have to preserve all the private interfaces. You want to be able to encapsulate behavior so that other objects can use the public interface and you can be free to change the implementation details.

Because there are no private methods in Smalltalk, any distinction between private and public methods has to be made using standards. One standard is to use the word `private` in the protocol names for private methods. You might see a protocol named `private` or `private-accessing`. The disadvantage of this scheme is that you don't always notice the protocol name. If you are browsing methods using senders, implementors, and messages, you are not told the protocol so may easily decide to use a particular message.

The technique I like is one described by Bob Brodd in The Smalltalk Report, Nov-Dec, 1994. All private methods are prefixed with `my`. Now you use common sense English to prevent the public use of private methods. Basically it looks strange to see something like `anObject myDoThis`, whereas it looks reasonable to say `self myDoThis`. If you ever see a `my` message sent to anything other than *self*, be suspicious — you are probably seeing a private method being used as if it were public. Here's an example of using the `my` prefix for private methods.

```
ImageCreator>>createImage
   self myRemoveSelfAsDependent.
   self myFileInApplication.
   self myRunStripper.
   self myOpenApplication.
   self myCloseLauncher.
   self myInstallEmergencyHandler.
   self mySaveImageAndQuit
```

If you use the `my` technique, there are two basic approaches you could take. One is to make everything public unless you know it is a support method that shouldn't be invoked in isolation. The other is to start by making all methods private, then making them public as you come across a message that needs to be sent by another object.

The advantage of the first approach is that you can't predict how other people will want to use your class, and if you make methods private by default, you don't give people the opportunity to be fully creative. The advantage of the second approach is that by making methods private by default, you define a public interface to your object and don't open up the implementation details to other objects. I generally use the latter philosphy because I want my objects tightly encapsulated. However, I also make the distinction between domain specific objects and general reusable objects; the latter usually have more public methods.

We can extend the concept of `my` methods to include *friend* methods. A friend message is one that can be sent between friend objects. Suppose you have a nicely encapsulated subsystem that consists of several collaborating objects. None of these objects has a life outside the subsystem, and they don't expect to receive messages from objects outside the subsystem. We can consider these objects all friends of each other, and define a prefix such as `fr` to denote messages that can only be sent by a friend. Each class that defines friend methods would have to provide documentation in the class comments describing exactly which classes are friends. Any programmer wanting to use a message starting with `fr` would have to look at the class comments to find out if the sending object is actually a friend.

In Chapter 29, Meta-Programming, we will show an extension to *Class* that automatically creates both public and private accessors for all our instance variables when we define a class.

Returning from a method

In Smalltalk, all methods return a value. There is no such thing as C++'s void function. By default all methods return the object the message was sent to. So, if there is not an explicit return in a method, in effect the last line of the method looks like:

```
^self
```

`^` is the return symbol and *self* refers to the message receiver — the object executing the method. Most methods implicitly return *self*. In the VisualWorks Browser, if you hold the *shift* key down while selecting a

method with the mouse you will see the decompiled code, which actually has a `^self` at the end. The return (`^`) is executed after all other message sends and assignments in the statement have been done.

Return object defined by method name

Often the method name will tell you what type of object will be returned. For example, method names that start with `is` will usually return a *Boolean*, *true* or *false*. For example, `isEmpty`, `isNil`. Messages that start with `as` will usually return the thing specified. For example, `asSortedCollection` sent to a collection will return a SortedCollection. The `asNumber` message sent to a string will return a number of the appropriate type. (However, `asString` is not implemented by numbers so you should use `printString` or `displayString` instead.)

Methods that add objects to collections, such as `add:` and `at:put:`, return the object that is added. Methods that remove single objects from collections, such as `remove:` and `removeFirst`, return the object that was removed. Because of this it's worth knowing about the `yourself` message. The following example illustrates how the use of `yourself` ensures that *collection* contains the list rather than the last object added (the example makes use of message cascading, which we will talk about in Chapter 6, Special Variables, Characters, and Symbols).

```
collection := List new
   add: thisObject;
   add: thatObject;
   yourself.
```

Consistency and predictability are great virtues, so when adding these types of methods to your own classes, follow the common usage patterns. Otherwise you'll confuse every programmer who looks at or uses your code.

Return object defined by last statement

If a method returns something other than self, the last line or the last statement in the method will specify what is being returned. The assumption when you see an explicit return at the end of the method is that the message sender will care about what is being returned. For this reason, don't do `^self` at the end of the method unless this has an explicit meaning for the sender.

Guard clauses

If you will execute the method body only if some condition is true, the obvious way to do this is:

```
MyClass>>someMethod
   (some condition) ifTrue:
      [self doThingOne.
      self doThingTwo.
      self doMoreLinesOfCode]
```

However, it is often better to return *self* from an `ifFalse:` block, then execute the main code. Thus we have a *guard clause* protecting entry to the method.

```
MyClass>>someMethod
   (some condition) ifFalse: [^self]
```

```
        self doThingOne.
        self doThingTwo.
        self doMoreLinesOfCode]
```

This makes the code a little easier to follow because it's all straight line code rather than conditional. This approach goes against the structured programming adage of one entry point, one exit point. The rule was created to solve the understandability problem of long functions with multiple entry and exit points. However, Smalltalk methods are usually short, so it's easy to see exactly what is happening.

Consistency in objects returned

If your methods implicity returns *self* at the end, use `^self` to return earlier. Don't return a different value, such as *nil*, unless *nil* has a specific meaning to the sender.

Misplaced Methods

If a method doesn't reference any instance variables, and simply operates on parameters, you might ask if the method is defined in the right place. If the method is manipulating a parameter, perhaps the code should be defined by the class of the parameter. For example, when working with a string, you may decide you need to remove the leading and trailing white space. So you write a new method for your class that trims the white space.

```
MyClass>>removeWhiteSpaceFrom: aString
   ... remove leading and trailing white space...
   ^newStringWithWhiteSpaceRemoved
```

However, this is a very procedural way of writing code. If you are thinking in terms of objects, you figure out where the responsibility really lies, and tell objects to do things. Since *MyClass* really is not responsible for removing white space from strings, where should that responsibility lie? The most obvious place is the class *String* (or one of its superclasses). You can then tell your string to strip itself of white space.

```
trimmedString := aString trimWhiteSpace.
```

Number of Methods

There is no rule for how many methods an object should have, but if you find that one object has many more methods than other objects, it may be that you are not distributing the workload well. In a good Object-Oriented system, you don't have super-intelligent objects. Instead, you have many peer-type objects cooperating. For more information, see Chapter 9, Object-Oriented Thinking.

Defaulting Parameters

In C++, you can leave out parameters in function calls and the parameters will be defaulted (assuming the function is written this way). In Smalltalk no such capabilitiy exists. Instead, you often see a method do nothing other than invoke another method with an additional parameter.

A good example of this is the `changed` message. When you invoke the dependency update mechanism, you use `changed` to specify that something has changed. You use `changed:` to specify what has changed, and `changed:with:` to specify the new value.

```
Object>>changed
   self changed: nil

Object>>changed: anAspect Symbol
   self changed: anAspectSymbol with: nil

Object>>changed: anAspectSymbol with: aParameter
   self myDependents
      update: anAspectSymbol
      with: aParameter from: self
```

Method Template

The method template (what you see when you are looking at methods but don't have a method selected) is defined in `Behavior>>sourceCodeTemplate`. If you don't like it, you can change it. For one example of how you might change it, see Chapter 31, Customizing your Environment.