Chapter

22 Common Errors

This chapter describes some of the more common errors that Smalltalk programmers make and gives suggestions on how to prevent the errors.

Notifier window messages

As you test your code you are likely to have exceptions raised. Here are a few exception messages you may run into, along with some text describing common causes of the exception.

```
Message not understood: #self (or some other object)
```

This occurs when you leave a period off the end of a statement. It tells you that the *self* with which you are starting a statement is being used as a message to the result of the previous statement. Ie, there is no statement separator. You may be sending the message to an object other than self, in which case that variable name will be the message name that was not understood.

```
Message not understood: #do:
```

This often occurs in a `printOn:` method, if the parameter to `nextPutAll:` is not a string. To correct this, send `printString` or `displayString` to the parameter, or use the `print:` message instead of `nextPutAll:`.

```
Message not understood: #startingAt:replaceElementsIn:from:to:
```

This often occurs when you are creating a string using the comma (`,`) message but the object on the right side of the comma is not of the appropriate class. For example, 'abc', 3 would generate this exception.

```
Message not understood: #someArbitraryMessage
```

The most common cause of getting a message not understood exception is to send the message to *nil*. Ie, your variable contains *nil* rather than the object you thought it contained.

Losing code you modified in a debugger

A common mistake when debugging is to add `self halt` to a method, and leave the method displayed in a Browser window. When you hit the halt, you step through the code, find the problem, and modify the code in the debugger. You then go back to the original window and make another change to the method. Since the Browser is still looking at the old version of the code, you lose the changes you made in the debugger. The easiest way to correct this problem is to deselect the method after adding a `self halt`. Then when you select it again, it has the changes.

Removing from a collection you are iterating over

If you are iterating over a collection using `do:`, and removing elements that satisfy some condition, you will probably get an exception. The invocation of `do:` sets up some boundary conditions which are no longer true when elements are removed because the collection will be rearranged as you remove elements. When I do the following, I get a "Message not understood" exception.

```
collection:= OrderedCollection withAll: #(1 2 3 4 5 6).
collection do: [ :each | each even ifTrue: [collection remove:
each]].
```

Instead, make a copy of the collection before iterating over it, such as shown below (of course, you should also consider using `select:` or `reject:`).

```
collection:= OrderedCollection withAll: #(1 2 3 4 5 6).
collection copy do: [ :each | each even ifTrue: [collection remove:
each]].
```

It's also possible to have the same problem even if you are not deleting objects directly. For example, the following code closes any open files with the specified path. As the files are closed, they are removed from the OpenStreams collection of ExternalStream, so again, we need to iterate over a copy of the collection.

```
(ExternalStream classPool at: #OpenStreams) copy
   do: [ :each | each name = aPath ifTrue: [each close]].
```

Not returning the injection variable in inject:into:

If you use `inject:into:`, you'll probably find that you sometimes forget to return the thing that is being injected. Remember that the value of a block is the value of the last statement executed, so you need to finish with something that guarantees the last thing is the injection variable. You can simply name the injection variable, or you can send it the `yourself` message. Here are examples of both.

```
self withAllSubclasses
   inject: OrderedCollection new
   into: [:coll :each | coll addAll: each allInstances; yourself]

#(3 $x 'hello' #mySymbol)
   inject: String new writeStream
   into: [ :stream :each |
      stream print: each class; nextPutAll: ' value '; print: each;
cr.
      stream]
```

Missing ^

A very common bug for new Smalltalk programmers is to forget the caret (^) when returning a value. This is usually seen when writing a `new` method. For example, the following creates a new, initialized instance of MyClass, then returns the class rather than the instance.

```
MyClass>>new
   super new initialize
```

The correct version should have a caret to return the instance. (It might also be worth using the `basicNew:` message — see Chapter 5, Instance Creation for more information.)

```
MyClass>>new
   ^super new initialize

MyClass>>new
   ^self basicNew initialize.
```

Not implementing =

If you create your own class then compare two instances of it using =, the test will return false. For example, two separately created instances of a new *Location* class will not compare as equal, even if all the details of the location are the same. This is because the default implementation of = is to use ==, so = is actually checking to see if they are the *same object*. If you create your own class and will be doing tests for equality between instances of the class, you need to write your own = method that compares the instance variables.

If you write your own = method, you will also need to write your own `hash` method, since two objects that are equal should also have the same hash value. (Hash values are used when putting objects in collections that use hashing, such as Set and Bag.)

Assuming that messages return self

A common bug is to assume that all methods return *self*, and in particular, to assume that `add:` returns *self*. The following will *not* give you what you expect, because `add:` returns the object added (similarly `remove:` returns the object removed, and `at:put:` returns the object put).

```
collection := OrderedCollection new
   add: objectOne;
   add: objectTwo.
```

The example above actually assigns objectTwo to collection. The following techniques all give you what you expected.

```
collection := OrderedCollection new.
collection
   add: objectOne;
   add: objectTwo.

(collection := OrderedCollection new)
   add: objectOne;
   add: objectTwo.
```

```
collection := OrderedCollection new
   add: objectOne;
   add: objectTwo;
   yourself.
```

The last example shows the use of the message `yourself`, which always returns the receiver. The use of `yourself` can make code a lot more robust. For example, both lines below should give you an employee object. However, the first line relies on the `salary:` accessor conforming to standards and returning *self.* The second line ensures that you always get the correct result.

```
employee := Employee new salary: aSalary.
employee := Employee new salary: aSalary; yourself.
```

Incorrect messages to Booleans

You send `ifTrue:`, `ifFalse:`, `ifTrue:ifFalse:`, and `ifFalse:ifTrue:` to the Booleans *true* and *false*. These values may be the result of evaluating an expression in parentheses. On the other hand, you send `whileTrue`, `whileFalse:`, `whileTrue:`, and `whileFalse:`, to instances of BlockClosure. Ie, the `if` messages are sent to a Boolean and `while` messages are sent to a BlockClosure. For example,

```
(some condition) ifTrue: [some code]..
(another condition)
   ifFalse: [this code]
   ifTrue: [that code].

[some code] whileTrue.
[some code] whileFalse: [more code].
```

Not reinitializing class variables

A common error when using class variables to contain static information is to modify the information, but not reinitialize the variable. Class side `initialize` methods are invoked when code is filed into the image, but once the code is in the image, the `initialize` method is not automatically sent again.

A common partial solution is to have a comment saying `"self initialize"` at the start of the `initialize` method, which you select and execute after accepting the modified method. For example,

```
MyClass class>>initialize
   "self initialize"
   Messages := Dictionary new.
   Messages at: #notFound put: 'Not found'.
```

Problems with copies

When you copy an object, the `copy` method does a *shallow* copy, which is a one-level-deep copy. The original object is copied, but no instance variables are copied — ie, the two objects have the same instance variable objects. Because of this, we have to be very careful what we do with copies. Let's look at examples of modifying a copied collection and a copied non-collection object. In the examples below, we'll use a *Person* class, with the instance variables *name* and *phone*, and accessors for both variables.

Modifying copy of non-collection object

Suppose you make a copy of an Employee object for safe keeping while you modify the original. If you do the following, you'll see that personTwo now has a phone number of '444-5555', which is not what you wanted.

```
personOne := Person new name: 'Alec'; phone: '555-5555'; yourself.
personTwo := personOne copy.
personOne phone replaceFrom: 1 to: 3 with: '444'.
personTwo inspect.
```

Now the chances are that you would not be replacing characters in a string, but rather the whole string, in which case there would be no problem. However, a Person object might contain an Address object. If you replaced the street address in personOne's address, you'd find this change reflected in personTwo. To overcome this problem, Person needs to implement `postCopy`. For more information, see Chapter 25, Hooks into the System.

Modifying copy of collection

Let's create a collection of Person objects then copy the collection. We might expect that the Person objects in the second collection are copies of the Person objects in the first collection. However, this is not the case. Even though we copied the collection, the objects in the copied collection are the *same* objects as the objects in the original collection. In this example, we modify the phone number of one person. If we inspect the copied collection, we'll see that the person has the new phone number, '222-2222'.

```
personOne := Person new name: 'Alec'; phone: '555-5555'; yourself.
personTwo := Person new name: 'Dave'; phone: '111-1111'; yourself.
collectionOne := OrderedCollection new add: personOne; add:
personTwo; yourself.
collectionTwo := collectionOne copy.
personOne phone: '222-2222'.
personTwo inspect.
```

If we create the second collection by copying the Person objects, we solve this problem. For example, if we create collectionTwo by doing the following, then inspect collectionTwo, the phone number will still be '111-1111'.

```
collectionTwo := collectionOne collect: [:each | each copy].
```

If you have an object that contains a collection in an instance variable, there's a question of whether you should even give access to the collection. For more information, see the section on Accessors for Collections in Chapter 4, Variables.

^ in block returns from method

There is unfortunately no way to return from a block other than hitting the right square bracket. If you use a caret (^) to return, you will exit the method as well as the block. For example, the following will exit the method.

```
MyClass>>myMethod
   someCondition ifTrue:
      [self someCode.
      anotherCondition ifFalse: [^false].
```

```
         self moreCode.
         self otherCode].
```

For the code to work correctly, you'll have to structure the code so that the right end of the block is hit. The first example keeps the code in the original method. The second example shows how you might put the block code in a different method to keep the methods simple and easy to understand.

```
MyClass>>myMethod
   someCondition ifTrue:
      [self someCode.
      anotherCondition ifTrue:
         [self moreCode.
         self otherCode]].

MyClass>>myMethod
   someCondition ifTrue:
      [self someCode.
      self checkConditionAndDoStuff].
```

Modifying a literal array

Modifying a literal array leads to unexpected behavior. For example, create a class *MyClass* and add the following method (the code can be found in the file `literal.st`).

```
MyClass>>methodOne
   | literalArray |
   literalArray := #(1 2 3 4 5).
   Transcript cr; show: literalArray printString.
   (literalArray at: 3) == 3
      ifTrue: [literalArray at: 3 put: 99]
      ifFalse: [Transcript cr; show: 'On no. It is already 99']
```

The first time you evaluate `MyClass new methodOne` you will see the following.

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

If you then evaluate the same line again, you will see this instead.

```
#(1 2 99 4 5)
On no. It is already 99
```

Looking at the method, it appears that you are assigning #(1 2 3 4 5) each time you execute the method. However, if you hold down the Shift key when selecting the method, you will see that the assignment line looks like the following.

```
MyClass>>methodOne
   | t1 |
   t1 := #(1 2 99 4 5).
```

In other words, the literal array has been modified in memory. Let's look at another example, this time using a WriteStream. We use the `with:` message to position the stream pointer to the end of the collection.

```
MyClass>>methodTwo
   | stream |
   stream := WriteStream with: #(1 2 3 4 5).
```

```
    Transcript cr; show: stream contents printString.
    stream nextPutAll: #(6 7 8 9).
    Transcript cr; show: stream contents printString
```

The first time we execute `methodTwo`, the Transcript shows:

```
#(1 2 3 4 5)
#(1 2 3 4 5 6 7 8 9)
```

The second time we execute the method, the Transcript shows the following.

```
#(1 2 3 4 5 6 7 8 9 nil nil)
#(1 2 3 4 5 6 7 8 9 nil nil 6 7 8 9)
```

Again, the literal array has been modified in memory. This time it has actually grown rather than just having element values changed. As before, you can hold the Shift key down while selecting `methodTwo` in the Browser, and you'll see that the literal array is not what you thought it was.

How do we prevent this problem? The best way is to never do things directly with a literal array in your code. Always do things with a *copy* of the literal array. In the above two examples, if we had written the lines as below, there would be no problem.

```
    literalArray := #(1 2 3 4 5) copy.
    stream := WriteStream with: #(1 2 3 4 5) copy.
```

So, for example, if you have a method that returns a literal string, have it instead return a copy of the string, as shown below.

```
MyClass>>localName
   ^'Request' copy
```