Chapter

18

Cleaning up at Termination

Having a garbage collector to take care of memory management for you is wonderful. Not having to worry about allocating and freeing memory makes life a lot easier and code a lot more bug-free. However, there are some things that the garbage collector does not take care of, and which you need to explicitly handle in the code.

In particular, external resources will stay open even if the objects that reference them are garbage collected. For example, if you open a file or a socket, it will stay open until you close it (or your Smalltalk image exits). Internally, if you have a Smalltalk process that stays in an infinite loop waiting for input to process, that process will never terminate, even if you no longer reference the process in your code.

Often you will open files, read or write them, then close them. Similarly, often you will fork Smalltalk processes that do some work then simply terminate. However, on other occasions you may open a file that you plan on keeping open for a while, such as a log file. You may have a Smalltalk process that sits in an infinite loop waiting for input from a socket or waiting for input from another Smalltalk process.

Exiting your image will close external files and sockets and terminate Smalltalk processes. However, when you are developing and debugging an application, bugs in your code will often raise an exception which you try to debug, then eventually terminate. This leaves processes running and files open. This chapter talks a little about how you might close files and terminate processes when you terminate the application in situations like this. For more information, see the article *Cleaning up after yourself*, by Alec Sharp and Dave Farmer, in the March-April, 1995 issue of The Smalltalk Report.

The CleanUp object

We define a new class called *CleanUp*, which tracks, for example, the sockets, files and Smalltalk processes that are permanently open. We don't add files that are opened, read, and then closed in a `valueNowOrOnUnwindDo:` block. Here's the definition of CleanUp, followed by the class initialization and instance creation code. This code can also be found in the file `cleanup.st`.

```
Object subclass: #CleanUp
   instanceVariableNames: 'processes files sockets '
   classVariableNames: 'AccessProtect '
   poolDictionaries: ''
```

```
   category: 'CleanUp'

CleanUp class>>initialize
   "self initialize"
   AccessProtect := Semaphore forMutualExclusion

CleanUp class>>new
   ^super new initialize
```

We create the mutual exclusion semaphore because our application may be running in many forked processes. To make sure that only one process is trying to add to the CleanUp object at a time, we use the semaphore to protect access to the various collections. On the instance side, we initialize the variables that hold on to the files, sockets, and processes, and we provide methods to add files, processes, and sockets to the CleanUp object. We also need methods to remove them from the object should the application code close them itself. We'll just show the code that adds files, since the code for processes and sockets is very similar.

```
CleanUp>>initialize
   processes := OrderedCollection new.
   files := OrderedCollection new.
   sockets := OrderedCollection new

CleanUp>>addFile: aFile
    ^AccessProtect critical: [files add: aFile]

CleanUp>>removeFile: aFile
   ^AccessProtect critical: [files remove: aFile]
```

When the application terminates, we want to close all the open files and sockets, and terminate all the processes. We nil the instance variables at the end so the garbage collector can recover the objects that were contained in the collections (we assume that a new CleanUp object will be created next time the application is run).

```
CleanUp>>doCleanUp
   AccessProtect
      critical:
         [processes do: [:each | each terminate].
         files do: [:each | each close].
         sockets do: [:each | each close].
         processes := files := sockets := nil]
```

Using the CleanUp object

Now that we have a CleanUp class defined, how do we use it in our application. The first decision is where to store it so that it is globally accessible. There's a lot more information on that topic in Chapter 7, Global Variables. For now, we'll just assume that it's stored in the class *MyGlobals* and is accessible through the `cleanUp` accessor. The next section of code shows (in a non-robust way) how we might add files, sockets, and processes to the CleanUp object in our code.

```
MyClass>>myStartInputOutput
   MyGlobals cleanUp addSocket: self myCreateInputSocket.
   MyGlobals cleanUp addSocket: self myCreateOutputSocket.
   MyGlobals cleanUp addProcess: [self myDoInputLoop] fork.
   MyGlobals cleanUp addProcess: [self myDoOutputLoop] fork.
```

```
MyClass>>myOpenLogFile
   logFile := self logFileName writeStream.
   MyGlobals cleanUp addFile: logFile
```

Finally, we need code to invoke the `cleanUp` message to the CleanUp object so that all files and sockets are closed and all processes are terminated. We do this by wrapping our main application start code in a `valueNowOrOnUnwindDo:` block. This makes sure that clean up is done whether the application terminated normally, we interrupted the application with ctrl-C, or we got an exception.

```
MyApplication>> start
   [self myStartInputOutput.
   [self myProcessObject: self myGetObject] repeat]
      valueNowOrOnUnwindDo: [MyGlobals cleanUp doCleanUp]
```

Other approaches

There are of course other approaches to the problem of making sure that you close and terminate everything that is open and running. The technique described in this chapter is a fairly easy and expedient technique. As you move toward a production system, a better and more object-oriented approach is to spread the responsibility to each subsystem or component. Each subsystem tracks what it has open or running. When the main application is shut down, it sends a `shutDown` message to each subsystem. The subsystem then shuts itself down gracefully, closing files, terminating processes, and doing any other appropriate operations.