Chapter

7
Global Variables

I'll start with the standard caveat that global variables should be used sparingly, if at all, and that most information should stored in instance variables or be passed as method arguments. Having said that, there is a definite use for global information and objects. You may have constants that are used through your system, or you may have global objects such as configuration objects or trace log objects. Let's look at some techniques that can be used for these purposes. We'll use two examples: a configuration object and a timeout value.

Global Dictionary

The most basic technique is to put global objects in the system dictionary named *Smalltalk* since objects in *Smalltalk* can be referenced from anywhere. For example, you might have code that does:

```
Smalltalk at: #Timeout put: 20.
Smalltalk at: #Config put: MyConfiguration new.
```

I recommend against using *Smalltalk* for application globals because it's used heavily by the Smalltalk system (for example, it contains all the classes in the system, and global variables such as *Transcript*, *Processor*, and *ScheduledControllers*). I prefer not to clutter *Smalltalk* up with application objects or risk name collisions.

Pool Dictionaries

The next technique is to use a *PoolDictionary*. A PoolDictionary contains objects to which you want global access for your application. The nice thing about a PoolDictionary is that classes have to register an interest in it before they can access its objects, so it allows you a certain amount of scoping. The way a class registers an interest is to name the PoolDictionary in the appropriate line in the class definition template. (In VisualWorks 2.0, subclasses inherit the PoolDictionary, but in VisualWorks 2.5, you have to explicitly name the PoolDictionary in each subclass that wants to reference PoolDictionary variables.) In the example below, I've called it MyPoolDictionary.

```
NameOfSuperclass subclass: #NameOfClass
   instanceVariableNames: 'instVarName1 instVarName2'
```

```
    classVariableNames: 'ClassVarName1 ClassVarName2'
    poolDictionaries: 'MyPoolDictionary'
    category: 'MyStuff'
```

Before you write code you must have done two things. First, before referencing the PoolDictionary in your class definition you have to make the name of the PoolDictionary universally known by adding it to the Smalltalk dictionary. To do this, execute the following. Note that we are using a Dictionary and not an IdentityDictionary. Because of the way an IdentityDictionary is implemented, it will not work as a PoolDictionary.

```
Smalltalk at: #MyPoolDictionary put: Dictionary new.
```

Before you write code referencing the objects in the PoolDictionary, you have to add these objects to the PoolDictionary so that the compiler can associate a variable name in your method with an object in the PoolDictionary. However, the compiler just needs to be able to reference the name in the PoolDictionary and doesn't care about the type of the object. For the purpose of accepting your method, the easiest thing to say initially is:

```
MyPoolDictionary at: #Config put: nil.
```

We don't want to do these steps manually every time we file in our code (see Chapter 33, Managing Source Code, for more information on managing projects and filing in code). So, at the beginning of the file that files in our code, we add the following.

```
Smalltalk at: #MyPoolDictionary put: Dictionary new.
MyPoolDictionary at: #Config put: nil.
```

If you have code that refers to PoolDictionary variables and you create a new PoolDictionary, the code will no longer be able to find the old variables. If you get an error in a method because a PoolDictionary variable is not found, and yet a PoolDictionary exists with the appropriate variable, recompile the method by making a small change and accepting the method. If this solves the problem, somewhere you created a new PoolDictionary while code still references the old one.

The question that always comes up when using a PoolDictionary is how do you find all the references to an object in your PoolDictionary, since you can't browse References To or Implementors Of? The answer is to evaluate the following code:

```
Browser browseAllCallsOn: (MyPoolDictionary associationAt: #Config).
```

It might be worth putting something like this in your Workspace and saving your image. VisualWorks 1.0 had a menu option to open the System Workspace which, among other useful code samples, had code to find PoolDictionary variable references. The System Workspace is not obviously available in VisualWorks 2.0 or 2.5, but you can get it by executing:

```
ComposedTextView openSystemWorkspace.
```

Advantages of using a PoolDictionary are: you refer to the objects in a PoolDictionary by a name starting with an uppercase letter, which makes it very obvious that the code references a global object; and, only classes with an interest in the PoolDictionary have access to it. Disadvantages of PoolDictionaries are: you have to take specific actions to allow PoolDictionary objects to be referenced, and it is harder to find all the references to the PoolDictionary object.

Class side variables

The next technique is to use a class to store global data. For example, we might have two classes: *MyGlobals* and *MyConstants*. Global and constant values are referenced by sending messages to the appropriate class. For example, we might reference and define a timeout value as follows.

```
MyClass>>myMethod
   timeout := MyConstants timeout.
   ...

MyConstants class>>timeout
   ^20
```

We might reference and initialize a global value (in this case a Log file) as fillows.

```
MyClass>>myMethod
   ...
   MyGlobals logfile log: 'salary=', salary printString.

MyGlobals class>>logFile
   ^LogFile

MyGlobals class>>initialize
   LogFile := MyLogFile on: self logFileName.
```

To find all references to an individual global, we can browse references to the message that returns the global. To see all the places that any global is used, we can browse class references for *MyGlobals*. Similarly, for constant values.

Unlike with variables in a PoolDictionary, we don't have to do anything special on fileIn, and we don't have to make sure that the reference contains an object when we write methods that refer to constants and globals. The only disadvantages of this approach over a *PoolDictionary* are that you can't restrict scope in the same way, and you have a class that does not act as a factory for instances.

Default instances

There's one other thing to talk about since we are on the topic of class side behavior. Sometimes you have a single instance of a class that you want to access from many places. Our Configuration object may be such an example. There's a definite temptation to say that there will never be more than one instance of this class, so we might as well just dispense with the instance and put all the information and behavior on the class side. That way you can access the behavior easily since class names are globally known.

Despite the temptation, it should be resisted. There's another way that allows similar ease of access without preventing you from creating additional instances. Suppose the class is *MyClass*. Define a class variable called

Default. On the class side, create a *class initialization* protocol with an `initialize` method [1]. In the class side `initialize` method do the following:

```
MyClass class>>initialize
    "self initialize"
    Default := self new.
```

You'll probably need to initalize the instance so you'll either need to write `new`, which invokes `initialize`, or in the class initalize method do `Default := self new initalize`. Then on the class side write a `default` method:

```
MyClass class>>default
    ^Default
```

In your application code, you can now write code such as:

```
length := MyClass default length.
```

If you want to get *really* fancy, you can assume that instance side messages being sent to the class are really intended for the default instance. You'd write the last example as:

```
length := MyClass length.
```

Given that the class doesn't understand the `length` message, how do we make this work? We can implement `length` on the class side as `^self default length`, or we can implement the method `doesNotUnderstand:` on the class side as follows.

```
MyClass class>>doesNotUnderstand: aMessage
   ^self default
      perform: aMessage selector
      withArguments: aMessage arguments
```

I don't particularly recommend this way of coding because it's not obvious how it's working, but hey, it's ideas like this that make Smalltalk so much fun. If you decide to use this technique, make sure your class comments contain explicit information about what you are doing.

Environment Variables / Command Line Arguments

Environment variables and command line arguments are not global variables, but since the information is globally available, and varies depending on the environment and how the image was invoked, we'll mention them here. The following returns an array containing the command line arguments. The first element is the program name and the second is the image name. Any additional array elements will be other command line arguments you added.

```
CEnvironment commandLine.
```

[1] Class initialization is only done on fileIn so it's a good idea to always put `self initialize` in comments at the start of class side `initalize` methods. That way you can reinitialize the class if you make changes to `initialize`. Of course, you can select the whole method and *doIt*, but having `self initialize` makes it more obvious that it may need to be done.

You can also retrieve environment variables using `getenv:`. For example,

```
CEnvironment getenv: 'LOGNAME'
```

Image Version Number

The version number for your image can be discovered in one of two ways. You can send the `version` message to Smalltalk, or the `versionId` message to ObjectMemory. For example, on a VW 2.0 system, I get the following.

```
Smalltalk version       'VisualWorks(R) Release 2.0 of 4 August 1994'
ObjectMemory versionId  #[42 26 35 0 20 0 0 4 42 26 35 4].
```

In particular, element 5 of the array returned by `versionId` is the image major version number. VW 2.0 gives 20, while VW 2.5 gives 25. So, to discover if you are using VW 2.0 you could do either of the following.

```
(Smalltalk version findString: '2.0' startingAt: 1) > 0
(ObjectMemory versionId at: 5) == 20
```