Chapter

29

Meta-Programming

In Smalltalk, we usually write code that creates objects then makes the objects interact by sending each other messages. Sometimes, however, you will want to do a different type of programming, one that makes use of the internal structure of classes and objects. You may want to find out what variables a class defines, what methods it implements, or whether it defines a particular protocol. You may want to do some operation for each subclass of a class. This type of programming is called meta-programming.

Meta programming is not for the faint of heart or for the inexperienced. However, as you become more comfortable with Smalltalk, you may find it useful to know something about how things are done behind the scenes. The main classes to look at are *Behavior*, *ClassDescription*, *Class*, and *Metaclass*, all of which reside in the Kernel-Classes category. We'll talk a little about how these classes interact, but for a fuller description see the Purple Book (*Smalltalk-80, The Language*, by Adele Goldberg and David Robson, Addison-Wesley).

Classes and Metaclasses

Let's start by taking a look classes and their metaclasses. Just as an OrderedCollection is an instance of the class OrderedCollection, the class OrderedCollection is an instance of another class. A class whose instances are classes is called a *metaclass*, so OrderedCollection is an instance of its metaclass. When you define a new class, a new metaclass is automatically created for it — the metaclass has exactly one instance, the class you created. The metaclass is an instance of the class *MetaClass* and it has no name, so you can't look at it in a Browser. If you inspect the two statements below that were evaluated in a plain VisualWorks 2.5 image, you'll see that the number of subclasses of Object is about twice the number of class names in Smalltalk [1]. This illustrates the point that each class has a unique metaclass.

```
Object allSubclasses size.              1903
Smalltalk classNames size.              960
```

[1] The first number is not exactly twice the second since there are a few classes that are not subclassed off Object. Also, the number of subclasses of Object is an odd number because Metaclass is a strange oddity, having a metaclass of Metaclass.

Since a class is an instance of its metaclass, what we think of as class side methods are really instance side methods of the class's metaclass. Every class has an instance variable, *methodDict*, that stores the methods that have been defined by the class for invocation by instances of the class. So in the methodDict variable of a class's metaclass are stored the methods that have been defined on the metaclass for invocation by its instances — in this case the single instance which is the class. Thus all methods are really instance methods. To see a concrete example for yourself, inspect `OrderedCollection` and look at the methodDict variable. This shows all the instance side methods of OrderedCollection. Now inspect `OrderedCollection class` and look at methodDict. You will see the class side methods of OrderedCollection. Figure 29-1 shows the two inspectors.

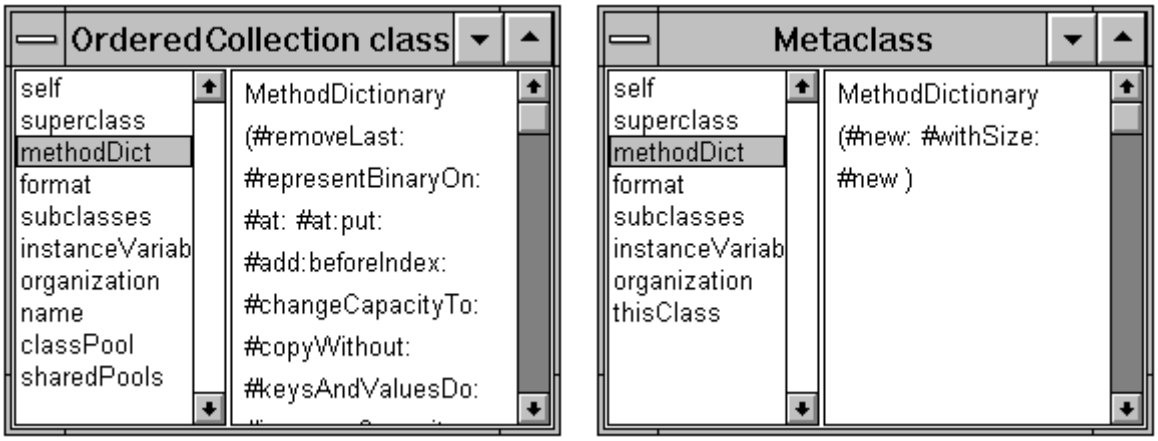

Figure 29-1. The methods of a class and its metaclass.

So, if you send `initialize` to an instance of a class, you are invoking the instance method of the class. If you send `initialize` to the class, you are invoking the instance method of the class's metaclass. For example, to count the number of methods defined for OrderedCollection on both the instance and class side, you can do the following. This counts the number of methods defined by OrderedCollection plus the number of methods defined by OrderedCollection's metaclass.

```
OrderedCollection selectors size + OrderedCollection class selectors
size.
```

Because class side methods are actually defined on the class's metaclass, the notation for class side methods of *MyClass class>>myClassMethod* is more than just a notation convention. It actually correctly describes the structure since `MyClass class` returns the metaclass.

Just as instance side methods are defined by the class and class side methods are defined by the metaclass, instance variables are defined by the class, and class instance variables are defined by the metaclass. A class has a variable named *instanceVariables*, in which are stored the names of the instance variables of the class's instances. Similarly, the class's metaclass stores the names of the instance variables of *its* instances — ie, of the class itself. To make all this talk of method dictionaries and instance variables clearer, take a look at the code in the file `metademo.st`. Figure 29-2 shows inspectors looking at instance variables of a class and its metaclass where the class has defined instance, class, and class instance variables.

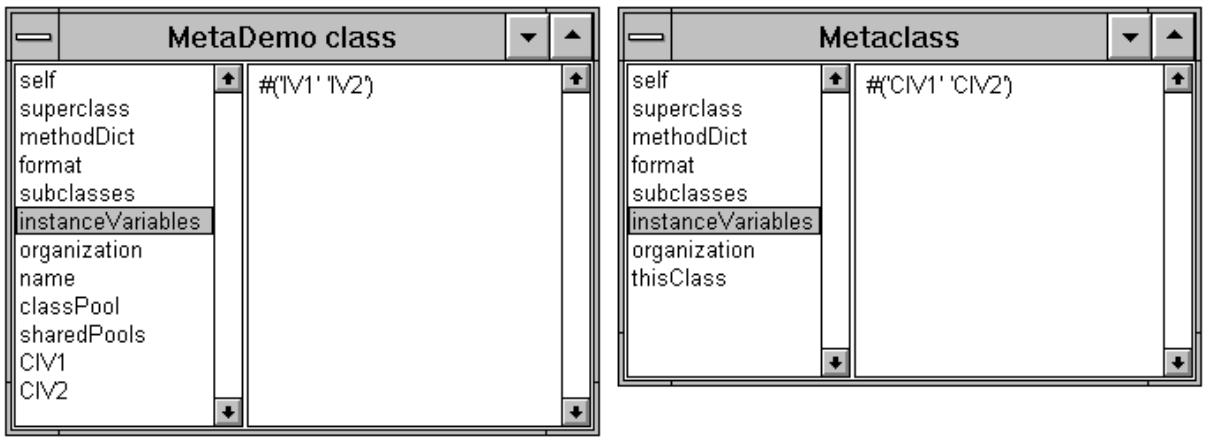

Figure 29-2. Instance variables of a class and its metaclass.

Adding general class side behavior

In chapter 32, Changes to System Classes, we will see examples of adding general instance side behavior to Object so that all instances inherit the behavior. In this section we will look at adding general class side behavior that is inherited by all classes. The obvious place to add general class methods is the class side of Object. However, if you look at the class side methods of Object you will see that there are very few of them, and that most of them are associated with the signals that Object keeps in its class variables. Methods such as `new` and `new:` are conspicuously absent. So if methods associated with class side behavior of subclasses is not found in Object, where are they found?

Just as instances execute methods defined either on their class or a superclass of their class, classes execute methods defined either on their metaclass or a superclass of their metaclass. So if you want a class side method that all classes will inherit, you can either write in on the class side of Object (where it is implemented on Object's metaclass), or on the instance side of one of the superclasses of Object's metaclass.

The superclass of Object's metaclass is *Class* — ie, `Object class superclass` is Class. Class has a superclass of *ClassDescription*, which has a superclass of *Behavior*. So to add general class side behavior, we can add it to the class side of Object or we can add it to the instance side of Class, ClassDescription, or Behavior. Both Class and ClassDescription are associated with the internal workings of classes and instances, with the handling of classes, variables, protocols, etc. Behavior provides additional behavior and is where `new` and `new:` are defined. The class comment for Behavior includes the following: "Most objects are created as instances of the more fully supported subclass, Class, but Behavior is a good starting point for providing instance-specific behavior." Thus, based on the purpose of these three classes, the instance side of Behavior is the most appropriate place to add general class side methods.

As an interesting note, the superclass of Behavior is Object. So, to add new class side behavior, we could actually add the method to the *instance* side of Object. We can demonstrate this by sending a method defined on the instance side of Object to a Class (for example, `OrderedCollection basicSize`). Try the following. Define the method `foo` on the instance side of Object, then send it to the class OrderedCollection and to an instance of OrderedCollection.

```
OrderedCollection new foo.
OrderedCollection foo.
```

```
Object>>foo
   Transcript cr; show: thisContext printString.
```

Of course, the problem with defining new class side behavior on the instance side of Object is that it will be inherited by both instances and classes, which is not what we want. Figure 29-2 illustrates the parallel instance and class side hierarchy.

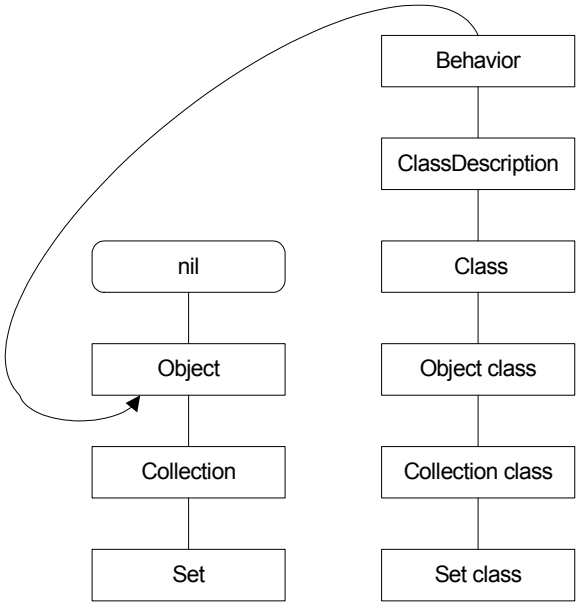

Figure 29-3.
The instance and class side hierarchy.

Examples we've seen in other chapters

We've seen examples of meta-programming in other chapters. For example, in Chapter 15, Printing Objects, we wrote `printAllOn:` using the messages `allInstVarNames`, `instVarIndexFor:` and `instVarAt:` to find the names and values of instance variables. In Chapter 30, Testing, we send the message `includesSelector:` to find out if a particular message selector is defined by a specified class. In the same chapter we send the messages `organization` and `listAtCategoryNamed:` to get the names of all the methods in a specified method protocol. In Chapter 33, Managing Source Code, we show how to count the number of classes and methods in the image using the `classNames` and `selectors` messages.

Examples we promised in other chapters

Finding classes with a specified protocol

In Chapter 20, Error Handling, we mentioned that we would write code to find all the classes that contain a specified protocol. If you follow the convention mentioned in Chapter 32, Changes to System Classes, you will put additions to system classes in a protocol named `(additions)`. When we get a new release of VisualWorks, we may need to find all the system classes to which we've added methods, so let's write our code to look for all classes that include the protocol `(additions)`.

```
Object allSubclasses select:
   [:each | each organization categories includes: #'(additions)'].
```

The above example finds all the classes that contain the `(additions)` protocol on either the class or instance side. It works because the subclasses of Object include both the regular classes and also their metaclasses. If we just wanted to find the classes that include the protocol on the class side we would look only at metaclasses, as in the example below. (Correspondingly, to find classes that include the protocol on the instance side we would replace `each isMeta` with `each isMeta not`.)

```
Object allSubclasses select:
   [:each | each isMeta and: [
      each organization categories includes: #'(additions)']].
```

The top-level classes in Smalltalk have *nil* as their superclass, with Object being the main top-level class. To find all the classes that have *nil* as their superclass, inspect `Class rootsOfTheWorld`. If you want to see *all* the classes that include the `(additions)` protocol, including Object and other class hierarchies that are subclassed off *nil*, do the following.

```
collection := OrderedCollection new.
Smalltalk allBehaviorsDo:
   [:each | (each organization categories includes: #'(additions)')
      ifTrue: [collection add: each]].
collection inspect.
```

The Browser makes heavy use of meta-programming, and you can find some interesting methods on its class side. For example, to find the names of all the methods in all the *class* side `(additions)` protocols, inspect the first example. To browse all the methods in these protocols, do the second example.

```
Browser allClassMethodsInProtocol: #'(additions)'.
Browser browseAllClassMethodsInProtocol: #'(additions)'.
```

To look at all methods rather than just class side methods, create a new method, `allMethodsInProtocol:`, which tells you all the classes with the specified protocol, regardless of whether the protocol appears on the class side or the instance side. Base the new method on `allClassMethodsInProtocol:` but remove the `isMeta` test. Also create a new method, `browseAllMethodsInProtocol:`, basing it on `browseAllClassMethodsInProtocol:`, but sending the `allMethodsInProtocol:` message instead of `allClassMethodsInProtocol`. You can now browse *all* your additions to system classes by doing the following.

```
Browser browseAllMethodsInProtocol: #'(additions)'.
```

allInstancesAndSubInstances

In Chapter 16, Processes, we mentioned that if you subclass off *Process*, you will need to modify code reading `Process allInstances` to now read `Process allInstancesAndSubInstances`. Here's the definition of this new method. Note the *yourself* in the `into:` block which guarantees that the block value is the collection that will be injected in the next iteration.

```
Behavior>>allInstancesAndSubInstances
   ^self withAllSubclasses
      inject: OrderedCollection new
```

```
      into:    [:coll :each | coll addAll: each allInstances;
yourself]
```

Creating accessors for instance variables

In Chapter 3, Methods, and Chapter 4, Variables, we mentioned that we would write code to automatically create accessors for our instance variables. To do this we add the following methods to *Class*. This mechanism will create public and private (`my` prefixes) accessors for each instance variable. You can then remove the ones that you don't need. To save space, we only show two of the methods that return the string used to create the accessing methods, and have compressed cascades and other messages. We also make use of the `capitalize` method we wrote in Chapter 12, Strings.

```
Class>>mySubclass: t instanceVariableNames: f classVariableNames: d
poolDictionaries: s category: cat
   | newClass |
   newClass := self
      subclass: t instanceVariableNames: f classVariableNames: d
      poolDictionaries: s category: cat.
   newClass instVarNames do:
      [:each |
      newClass compile: (self getAccessor: each) classified:
#accessing.
      newClass compile: (self setAccessor: each) classified:
#accessing.
      newClass compile: (self myGetAccessor: each) classified:
#'private-accessing'.
      newClass compile: (self mySetAccessor: each) classified:
#'private-accessing'].
   ^newClass

Class>>getAccessor: aName
   | stream |
   stream := String new writeStream.
   stream nextPutAll: aName.
   ^self getAccessor: aName stream: stream

Class>>mySetAccessor: aName
   | stream |
   stream := String new writeStream.
   stream nextPutAll: 'my'.
   stream nextPutAll: aName capitalize.
   ^self setAccessor: aName stream: stream

Class>>getAccessor: aVariableName stream: aStream
   ^aStream crtab; nextPut: $^; nextPutAll: aVariableName;
   contents

Class>>setAccessor: aVariableName stream: aStream
   | value prefix |
   value := aVariableName capitalize.
   prefix := (value at: 1) isVowel ifTrue: ['an'] ifFalse: ['a'].
   value := prefix, value.
   ^aStream
      nextPutAll: ': '; nextPutAll: value; crtab; nextPutAll:
aVariableName;
      nextPutAll: ' := '; nextPutAll: value; contents
```

To use this mechanism, when you define a new class, replace the `subclass:` keyword with `mySubclass:`. When you accept the class definition, the new protocols will be created and public and private accessors will be created for every instance variable. For example, the first two lines of the class definition might look something like the code shown below. Following is an example of one of the accessors that was created for *age*.

```
Object mySubclass: Person
   instanceVariableNames: 'name age address '

Person>>myAge: anAge
   age := anAge
```

A brief overview of other capabilities

There are a wealth of methods that do interesting things. You can find all the immediate subclasses of a class (`subclasses`), the whole hierarchy of subclasses (`allSubclasses`), the class itself with all it subclass hierarchy (`withAllSubclasses`), and the corresponding superclass methods (`superclass`, `allSuperclasses`, and `withAllSuperclasses`).

You can see if a class is identically equal to another class or is a subclass of it (`includesBehavior:`), if it is simply a subclass (`inheritsFrom:`), or if it is an immediate subclass (`isDirectSubclassOf:`).

You can look at all the instances of a particular class (`allInstances`), all the objects that reference a particular object (`allOwners` and `allOwnersWeakly:`). (If you do either of these, you might want to do `ObjectMemory garbageCollect` beforehand, to eliminate any unused objects that are waiting to be garbage collected.) You can iterate over all the subclasses of a class (`allSubclassesDo:`), all the instances of a class (`allInstancesDo:`), and all the instances of a class's subclasses (`allSubInstancesDo:`).

You can look at all the methods a class defines (`selectors`), and all the methods it understands, including inherited methods (`allSelectors`). You can see if an object understands a particular message (`respondsTo:`) and whether a particular method is defined by a class (`includesSelector:`). You can even find which class in the hierarchy defines a method that your object responds to (`whichClassIncludesSelector:`).

You can look at all the instance variable names defined by a class (`instVarNames`), all the instance variable names that it both defines and inherits (`allInstVarNames`), and similarly for class variable names. You can find the number of instance variables (`instSize`), the index for a particular instance variable name (`instVarIndexFor:`) and the value of an instance variable for a given index (`instVarAt:`).

Tread lightly when doing programming that involves knowledge of class structure. It can be useful, but should not be used as a substitute for good application design.