Chapter

5
Instance Creation

We've now covered enough material to look more closely at creating instances of a class. The basic instance creation message is new, which returns a new instance of the class. For example,

```
employee := Employee new.
collection := OrderedCollection new.
```

If you send new to a collection class, it will return a new collection[1]. The size will always be zero since the collection is empty, but the capacity will be the default capacity for that class. The capacity is the number of items you can put in the collection before it has to grow. For example, printing the following expressions gives the capacity shown on the right.

```
Array new capacity.                              0
Bag new capacity.                                0
Dictionary new capacity.                         3
Set new capacity.                                3
List new capacity.                               5
OrderedCollection new capacity.                  5
```

Growing a collection is quite expensive because the growing code creates a new collection with a larger capacity, copies over all the elements of the collection, then becomes the new collection. If you know how big the collection should be, or have an idea for the starting capacity, it's more efficient to create the collection by sending the message new: with the starting capacity as a parameter. Some collections, such as instances of *Array*, don't grow automatically, so it's important to specify their capacity (for collections that don't automatically grow, the size and capacity are the same). For example,

```
array := Array new: 10.
collection := OrderedCollection new: 20.
stream := (String new: 100) writeStream.
```

The new and new: methods are implemented by *Behavior*, although they are overridden by many other classes. Because your class will inherit new and new:, you don't need to override them unless your class has data that needs to be set when an instance is created. There are two types of data setting that can be done: setting variables to a default initialized value, and setting variables to data that is specific to the instance. Let's take a look at each of these.

Setting default values

Suppose we have an *Employee* object that tracks the number of hours of vacation and sick time allowed and taken. When an instance of *Employee* is created, we want to initialize these instance variables to the appropriate values, regardless of the name or salary of the employee. There are two ways of doing this. We can use lazy initialization as we saw in Chapter 4, Variables. For example,

```
Employee>>sickHoursUsed
   ^sickHoursUsed isNil
      ifTrue: [sickHoursUsed := 0]
      ifFalse: [sickHoursUsed]
```

Alternatively, we could initialize all the data in a single `initialize` method. Lazy initialization is useful if you have objects where the data may never be accessed and where it's expensive to initialize the data, but it has a cost of an extra message send each time the data is accessed. If you will be accessing the data a lot, it's worth doing the initialization in an `initialize` method. For example, for our *Employee* object we might have the following (although we wouldn't have hard coded values for the *allowed* variables).

```
Employee>>initialize
   self sickHoursUsed: 0.
   self vacationHoursUsed: 0.
   self sickHoursAllowed: 80.
   self vacationHoursAllowed: 80.
```

To invoke the `initialize` method you'll have to override `new` since the inherited `new` doesn't invoke `initialize` (at least if *Employee* is subclassed off *Object*). The usual way to override `new` is as follows (one of the most common errors of beginning Smalltalk programmers is to leave off the ^, which means that the class itself will be returned, rather than the newly created instance).

```
Employee class>>new
   ^super new initialize
```

Before you override `new` like this, you need to be aware of what `super new` does. If the `new` method in the superclass sends `initialize`, your `initialize` method will be invoked twice, first by the superclass `new` method, then by the *Employee* `new` method. In this situation you don't need to override `new` since you can inherit it from your superclass. Since the superclass has an `initialize` method that presumably initializes superclass data, your `initialize` method should start with `super initialize`. For example, suppose we have an *Employee* class, with subclasses of *HourlyEmployee* and *SalariedEmployee*. Let's assume that hourly employees get two weeks of vacation while salaried employees get three weeks. We might have the following:

```
Employee class>>new
   ^super new initialize
```

[1] Sending `new:` to a class that is not variable sized will generate an exception.

```
Employee>>initialize
   self sickHoursUsed: 0.
   self vacationHoursUsed: 0.

HourlyEmployee>>initialize
   super initialize.
   self sickHoursAllowed: 80.
   self vacationHoursAllowed: 80.

SalariedEmployee>>initialize
   super initialize.
   self sickHoursAllowed: 120.
   self vacationHoursAllowed: 120.
```

While overriding `new` to be `^super new initialize` is the common way of doing it, some people prefer to use the `basicNew` message.

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

Methods that start with `basic`, such as `basicNew` and `basicAt:`, should not be overridden. Their purpose is to provide the basic functionality, and programmers should be able to rely on this. If you want to override the functionality, override `new` and `at:`. By using `basicNew`, you don't have to worry about any superclass sending `initialize` and thus causing your `initialize` method to be invoked more than once. However, you still need to determine whether you should send `super initialize` in your `initialize` method.

Overriding new

It gets frustrating to have to override `new` just so you can invoke `initialize`. One solution is to have all your application classes subclassed of *MyApplicationObject*, which is subclassed off *Object*. In MyApplicationObject, you override `new` on the class side, and write a default `initialize` on the instance side. Now you can override `initialize` in your class without having to override `new`.

```
MyApplicationObject class>>new
   ^self basicNew initialize

MyApplicationObject >>initialize
   "do nothing"
```

Setting instance specific values

Often when you create a new instance you want to give it some information. In our employee example, we need at least a name. We may also need to provide a social security number, a salary, and information about gender, marital status, and number of dependents. There are two choices: to create the instance then set the variables, and to set the variables as part of instance creation. For the sake of example, let's assume that when creating an employee object, two pieces of information are absolutely required: a name and a social security number. If we create the instance then set the variables, we might have something like:

```
employee := Employee new.
employee name: aName.
employee socialSecurityNo: aSocialSecurityNo.
```

The problem with this approach is that you are relying on all programmers to remember to set the required variables after creating the instance. This is okay if the variables are optional, but dangerous if they are required. If you need to guarantee that the data is set, you are better off writing a instance creation method that forces programmers to provide the required information. For example, if we write our own instance creation method, we can create an employee like this:

```
employee := Employee name: aName socialSecurityNo: aSocialSecurityNo.
```

What would the `name:socialSecurityNo:` method look like? One option would be to simply pass on to an initialization method the information that needs to be set.

```
Employee class>>name: aName socialSecurityNo: aSocialSecurityNo
   ^super new initializeName: aName socialSecurityNo:
aSocialSecurityNo
```

This is a reasonable approach if you need an initialization method to initialize other data, such as the vacationHoursUsed variable shown above. However, if the initialization method does nothing except set the variables passed in, you might set the data directly. For example, you could use one of the following techniques; the second one dispenses with the temporary variable.

```
Employee class>>name: aName socialSecurityNo: aSocialSecurityNo
   | instance |
   instance := super new.
   instance name: aName.
   instance socialSecurityNo: aSocialSecurityNo.
   ^instance

Employee class>>name: aName socialSecurityNo: aSocialSecurityNo
   ^super new
      name: aName;
      socialSecurityNo: aSocialSecurityNo;
      yourself
```

Overriding new to avoid it being used

If you require programmers to use `name:socialSecurityNo:` to create instances of *Employee*, you could override `new` to raise an exception. Doing this is not very common, but it does make it easier for programmers to discover that they are creating employee objects in the wrong way.

```
Employee class>>new
   self error: 'Please use name:socialSecurityNo: to create Employee
instances'
```

Avoiding the use of new:

If only the employee name is required, you might be tempted to use `new: aName`. Resist the temptation. The instance creation message `new:` is used to specify the size of a collection, and programmers reading code should be able to assume that a collection is being created when they see `new:`. Instead, use `name:` or `newNamed:` or `newWithName:`. I tend to like method names that tell me both that they are creating a new instance and what the parameter is.

Sole instances of a class

Some classes have but a single instance. Examples in the system classes are *true*, which is the sole instance of *True*, *false*, which is the sole instance of *False*, *nil*, which is the sole instance of *UndefinedObject*, and *Processor*, which is the sole instance of *ProcessorScheduler*. The classes *UndefinedObject*, *Boolean*, and *ProcessorScheduler* override `new` to prevent new instances being created.

In your own code, if you have a class that should have only one instance, the easiest way to handle this is to have a class variable that contains the sole instance. When someone tries to create a new instance after the first one, you can either raise an error or return the sole instance. For example,

```
MyClass class>>new
   Instance isNil ifFalse: [self error: 'You can only have one
instance of MyClass'].
   Instance := self basicNew.
   ^Instance

MyClass class>>new
   ^Instance isNil
      ifTrue: [Instance := self basicNew]
      ifFalse: [Instance]
```