Chapter

4

Variables

In this chapter we'll look at the different types of variable that are available in Smalltalk: instance, class, class instance, parameter, and temporary. Global variables are a big enough topic in themselves that we cover them in Chapter 7, Global Variables. Smalltalk also has a few special variables that we will see in Chapter 6, Special Variables, Characters, and Symbols.

Variable names

Code is easier to understand when all method names and variable names are meaningful and easy to understand. Make variable names explicit and obvious so that a reader immediately knows the purpose of the variable. Don't abbreviate names; spell them out in full unless there are abbreviations that are accepted by all programmers and are part of the project vocabulary. Your goal should be that a programmer can look at your code and immediately understand what is going on, even without comments in the code. Part of this involves a good division of responsibility in the code and part involves well thought out names for methods and variables.

Variable names consist of one or more words strung together. The first letter of each word should be capitalized, and the rest of the letters should be lowercase. The exception to this is the first letter of the name for which the following rule applies: Instance, parameter, and temporary variables should start with a lowercase letter. Class, class instance, and global variables should start with an uppercase letter. Some people leave acronyms as uppercase; I try to avoid having two capital letters in a row as it makes it just that little bit harder to break the name into words. Here are some examples of variable names.

```
"Instance or temporary variables"
employeeName
collectionOfStrings
upperLeftCornerOfBox

"Parameter"
aNumber
aCollectionOfEmployees

"Class, Class instance, or Global variables"
Employee
```

```
DaysInMonth
MonthNames
```

I've always found it useful to prefix class names with an application or component prefix because I like to know where a class comes from when I see it in the code. For example, if you have an accounting application with payroll, accounts payable, and general ledger components, you might have prefixes such as `Pr`, `Ap`, `Gl` for the components, then something like `Aac` for the classes used for the communication between components (Accounting Application Communication). General support classes that will be used across applications or components get a general company prefix, or a prefix such as `Sup`.

Instance variables

Suppose we have a class called *Friend*. Each friend in our application has a first name, a last name, and a phone number. The class definition for *Friend* specifies that each instance of *Friend* has three variables: *firstName*, *lastName*, and *phone*. We specify these *instance variables* in the class definition.

```
Object subclass: #Friend
   instanceVariableNames: 'firstName lastName phone'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'My-Category'
```

When we create a new instance of *Friend*, it will have its own values in the instance variables. We might create an instance of *Friend* by sending the following instance creation message to the class *Friend*.

```
friendOne := Friend
   newLastName: 'Doe'
   firstName: 'John'
   phone: '555-5555'
```

If we then create another friend, friendTwo, it has no effect on friendOne, because friendOne has its own values in the instance variables.

```
friendTwo := Friend
   newLastName: 'Smith'
   firstName: 'Jane'
   phone: '111-1212'
```

The two instances of *Friend* have their own values in the instance variables, independent of the values in the other instance. If we change the value of one instance variable, it has no effect on the other friend. For example, we can change the phone number for John Doe without affecting the phone number of Jane Smith.

Class variables

Suppose we have several instances of class *Date*. Each instance will be unique, and will have its own values for the instance variables defined by *Date* (these are *year*, the year, and *day*, the number of days since the beginning of the year). We want to know the name of the month for each of these instances. Something has to keep track of the names of the months, and since month names are associated with dates, we might as keep track of the names somewhere in the *Date* class. However, we don't need to waste space by having each instance of a *Date* keep this array. Instead, we put the array of month names in a *class variable*. This lets the *Date* class keep

track of the array, so we only have one copy of the array. All instances have access to the class variables of their class, so each instance of *Date* can access the month name array. In fact, *Date* has many class variables, as shown in the following class definition.

```
Magnitude subclass: #Date
   instanceVariableNames: 'day year '
   classVariableNames: 'DaysInMonth FirstDayOfMonth MonthNames
SecondsInDay WeekDayNames '
   poolDictionaries: ''
   category: 'Magnitude-General'
```

Here are a two examples where instances of *Date* reference some of the class side variables. The messages `monthName` and `daysInMonth` look at the arrays held in the class variables *MonthNames* and *DaysInMonth*.

```
(Date newDay: 115 year: 1960) monthName.       #April
Date today monthName.                          #October
Date today daysInMonth.                              31
```

All instances can directly reference their class variables, although it may be preferable to reference them through *accessors*, which we'll look at later in this chapter. Although instances have access to the class variables of their class, the class itself does not have access to any instance variables of its instances, and in fact, the class doesn't usually even know that it has instances. You can inspect the class variables of a class by inspecting the class then inspecting the *classPool* variable[1].

Classes referencing their instances (advanced)

Despite the fact that classes don't by default know about their instances, you will sometimes find that classes manage their instances. You may also find that there is a concept of the *current* instance of a class. Combining these concepts (which doesn't usually happen), you might see something like the following in a class definition. Note that what we describe here is very simplistic and is just designed to point out the concepts.

```
Object subclass: #MyClass
   instanceVariableNames: 'instVar1 instVar2 '
   classVariableNames: 'Instances Current '
   poolDictionaries: ''
   category: 'MyCategory'
```

When a new instance is created, it is stored in a collection of instances held by the *Instances* class variable. There is also a method to retrieve and set the current instance so that the rest of the system knows which instance of *MyClass* they should be working with.

```
MyClass class>>initialize
   "self initialize"
   Instances := OrderedCollection new.
```

[1] Alternatively, you can send the `classPool` message to the class (eg, `Date classPool`). If you know the name of the class variable, you can do something like `Date classPool at: #DaysInMonth`. (Interestingly, you can inspect the class variables of Object directly, just by typing the variable name in a text window such as a workspace or a Browser, then inspecting the name. This works because in a text window, *self* is either *nil* or the class that you are browsing. Because of inheritance, both of these objects have direct access to the class variables of their superclasses, and so can see the class variables of Object.)

```
MyClass class>>new
   "This shows the concepts"
   newObject := super new initialize.
   Instances add: newObject.
   ^newObject

MyClass class>>new
   "This shows a tighter implementation"
   ^Instances add: super new initialize.

MyClass class>>current
   ^Current

MyClass class>>current: anInstance
   "Set the current instance and inform dependents, telling them the
old value"
   old := Current.
   Current := anInstance.
   self changed: #current with: old
```

This last method shows some code associated with the Smalltalk dependency mechanism. We'll see more about this in Chapter 19, The Dependency Mechanism, but for the time being we'll simply remark that it provides a way for other objects to be told when the current instance has changed and gives them a chance to take whatever actions are meaningful for them as they change to the new current instance.

Class instance variables

Besides class variables and instance variables, Smalltalk provides a variable called a *class instance variable*. It's a variable defined on the *class* side, and provides a way to hold a value that is potentially different for each subclass of the original class. Instances of each subclass all have access to the variable, but instances of one subclass will see a different value than will instances of another subclass.

It's not very common to see examples of class instance variables, but they can be useful. Here's an example of one such use. Suppose that your application has a centralized error message facility (we talk more about this in Chapter 20, Error Handling). However, each component of the application wants to have its own error messages and its own numbers, each prefixed by the appropriate tag. By having separate subclasses for the different components, programmers can work on the components in isolation without having to worry about message symbol or number conflicts. The component error messages classes are subclassed off *ErrorMessages*. Note that all the definitions and methods below are on the class side.

```
ErrorMessages class
   instanceVariableNames: 'Messages'

ErrorMessages class>>initialize
   Messages := Dictionary new.

ErrorMessages class>>number: aSymbol
   ^(Messages at: aSymbol ifAbsent: [self notFoundError: aSymbol])
key printString

ErrorMessages class>>notFoundError: aSymbol
   ^0->('Symbol <', aSymbol, '> not found')

ComponentOneErrorMessages class>>initialize
```

```
   super initialize.
   Messages
      at: #notFound    put: 1 -> 'Account not found';
      at: #duplicate   put: 2 -> 'Duplicate account'.

ComponentOneErrorMessages class>>number: aSymbol
   ^'C1-', (super number: aSymbol)

ComponentTwoErrorMessages class>>initialize
   super initialize.
   Messages
      at: #duplicate   put: 1 -> 'File already exists';
      at: #notFound    put: 2 -> 'File not found'.

ComponentTwoErrorMessages class>>number: aSymbol
   ^'C2-', (super number: aSymbol)
```

To sum up, class instance variables are used when you need a class variable to store information that may be different for each subclass. We show another use of class instance variables in Chapter 30, Testing, where we use them to hold the display name of the test case subclasses.

Parameters

Parameters (or arguments) are objects that are passed into a method or a code block. Smalltalk doesn't have type checking, so you can pass any object as a parameter to any method. Of course, when you send a message to the parameter object, there will be problems if the object doesn't understand the message — if the object is not of the appropriate type. (Some people jokingly say that Smalltalk is very strongly typed — all arguments must be objects!) Unlike the other types of variable, you can't assign a different value to a parameter variable. Fortunately, the compiler will catch this, so you won't even be able to accept the following.

```
MyClass>>doSomethingWith: anObject
   anObject := 3.
```

However, you can modify the contents of a parameter, so it's perfectly legitimate to do the following, although not necessarily good style.

```
MyClass>>modifyCollection: aCollection
   aCollection add: 3.
```

Method parameters

In the code shown above, we've seen various examples of messages and parameters. Shown below are a few examples of method and parameter names as shown in the various definitions.

```
Date class>>newDay: dayCount year: referenceYear
Object>>changed: anAspectSymbol with: aParameter
OrderedCollection>add: newObject
Dictionary>>at: key put: anObject
MyClass>>modifyCollection: aCollection
```

The objects after the colons are the parameters. Why do the parameters have these names? There are two types of naming scheme: you can name the parameter according to its *content* (eg, *lastName, salary, price, quantityOnHand*), or according to its *type* (eg, *anInteger, anArray, aSymbol*) As a general rule you'll find

instance and temporary variables named according their content, while method parameters are named according their type. You'll often see methods that have parameters with names such as *aString*, *aCollection*, *anEmployee*, *aRobot*. This type information tells the programmer a lot about the messages that can be sent to the object. Sometimes, however, *type* information is not enough.

In the examples above, the parameter to `newDay:` could have been *anInteger*, but this would provide very little information about what the content of the integer should be. So, the parameter is named *dayCount*. Similarly, we might have a method that creates a new instance of *Employee* as follows.

```
Employee class>>firstName: firstName lastName: lastName
```

It makes more sense to give the parameters names such as *firstName* and *lastName* rather than *aStringOne* and *aStringTwo*. Even names such as *aFirstNameString* don't add a lot of information, especially since the method will probably not do anything with the parameters other than store the values in instance variables.

To summarize, method parameters are often named according to their type. However, there will be situations where naming them according to their content makes more sense, and situations where naming them according to a mixture of type and content makes sense.

Block parameters

Code blocks contain code that is not executed until the block is sent a message from the `value` family. To pass a parameter to a block, the message will have to be one that has a parameter, such as `value:` or `value:value:`. For example, we might have something like the following.

```
block := [ :nameString |Transcript cr; show: 'The name is ',
nameString ].
block value: 'Alec'.

block := [ :nameString :age |
         Transcript cr; show: 'The age of ', nameString, ' is ', age
printString].
block value: 'Dave' value: 12.
```

You probably won't often use blocks with more than one or two parameters, but if you do, you can send `value:value:value:` for blocks with three parameters, or `valueWithArguments: anArray` if you have more than three. For example, a block with five parameters might look like the following. If the number of array elements doesn't match the number of parameters, you'll get an exception.

```
[ :parm1 :parm2 :parm3 :parm4 :parm5 | self doSomething ]
   valueWithArguments: #(99 88 77 66 55).
```

You'll see block parameters used in the enumeration methods of collections. For example,

```
aCollection do: [ :each | Transcript cr; show: each printString].
aCollectionOfNumbers collect: [ :each | each * 30].
aCollectionOfNumbers inject: 0 into: [ :subtotal :each | subtotal +
each ].
```

When naming the parameters of a block that is used when iterating over a collection, one convention is to use the parameter name *each*. This tells you instantly that the variable is the current element of the collection.

Alternatively, use the name *index* if you know that the variable is an index, or the name *char* if the variable is a character in a string.

```
1 to: 5 do: [ :index | self doSomethingUsing: index ].
'now is the time' do: [ :char | self doSomethingUsing: char ].
```

Generally, the only time I would use a name other than *each, index* or *char* for the collection element is if there are nested collection iterations, and I can't use the same name in both of them. For example,

```
#('cat' 'dog' 'gerbil')
   collect: [ :pet | #('milk' 'water' 'oj')
      collect: [ :drink | pet -> drink]]
```

Temporary variables

Temporary variables are variables that exist only for a short time: for the duration of a method or the duration of a block of code. There are several reasons to use a temporary variable. The most important reason is to *capture a value that can't be regenerated*. For example, if you are reading an object from a stream or a shared queue, and you want to use that object several times, use a temporary variable to capture it. For example:

```
request := sharedQueue next.
originator := request originator.
requestTime := request creationTime.
```

Another reason to use a temporary variable is to *avoid having to repeat expensive operations*. For example, if you are comparing a variable with a value from a database, it's appropriate to store the database value in a temporary variable. Compare the two examples below.

```
aCollection detect: [ :each | each = self myValueFromDatabase ]
ifNone: [nil].

databaseValue := self myValueFromDatabase.
aCollection detect: [ :each | each = databaseValue ] ifNone: [nil].
```

A third reason to use a temporary variable is to *increase performance by reducing the number of message sends*. For example, in the first example below, we have to do several identical message sends several times. It saves a lot of message sends if we do it the way shown in the second example.

```
self checkAddress: (employeeCollection at: employeeId) address.
self validateSalary: (employeeCollection at: employeeId) salary.
self printCheck: (employeeCollection at: employeeId).

employee := (employeeCollection at: employeeId).
self checkAddress: employee address.
self validateSalary: employee salary.
self printCheck: employee.
```

A fourth reason to use a temporary variable is to *make it easier to understand the code*. Sometimes it can be difficult to understand what object we have as a result of a complex sequence of message sends. While we may not need to use a temporary variable if we only do this sequence of message sends once, it can make the code more readable to store the result of this sequence in a well named temporary variable. (An alternative approach is to compute the complex result in another method and replace the complex message sends with a single message

send.) For example, the code below makes it easy to see what is going on without having to read through the sort block code.

```
MyClass>>mySortByBirthday: aCollection
   | birthdaySortBlock |
   birthdaySortBlock := [ :first :second | first birthday <= second
birthday ].
   ^aCollection asSortedCollection: birthdaySortBlock.
```

Temporary to the method

Temporary variables are named between vertical bars before any code in the method. For example,

```
copyWith: newElement
   | newCollection |
   newCollection := self copy.
   newCollection add: newElement.
   ^newCollection
```

You can type in the temporary variable name between the vertical bars, or you can let the compiler generate the name between the bars. The compiler will generate the names of temporary variables in the order in which they occur in the method. I usually let the compiler generate the names, but this sometimes leads to problems. If you give a temporary the same name as an instance variable, the compiler will assume that you are referring to the instance variable and not generate the name for you. (If you type the name in yourself, you will be warned that a variable of that name already exists, perhaps in an outer scope). On the class side, if you name a temporary variable *name* and don't put it between vertical bars yourself, the class itself will be given a new name when the code is run!

Temporary to the block

Temporary variables in blocks are enclosed between vertical bars, just as method temporaries are. For example,

```
[ | result |
result := self myDoSomething.
... ]
```

If you have block parameters and block temporary variables, they are defined as follows. Note that a vertical bar appears after the parameters, and the temporary variables are defined between their own vertical bars. So there are two vertical bars.

```
myMethod
   aCollection do: [ :each |   | result |
         ....
```

BlockClosures have better performance if all the variables they reference are local to the block. So, rather than letting the compiler generate temporary variable definitions at the top of the method, look to see if it is possible to make all block references internal to the block.

Global variables

Chapter 7, Global Variables, covers global variables, so we won't discuss them here.

Variables as slots

Variables are simply slots which hold an object. The object itself exists somewhere in computer memory, but it is *bound* to the variable. If we bind a different object to the variable, the first object may no longer be bound to any variable. When an object is not bound to any variable — ie, it is no longer referenced, the object is available to be garbage collected. In concept it is floating somewhere in memory with nothing attached to it, and the garbage collector comes along and sweeps it up.

Because an object exists separately and is just bound to a variable, there's no reason that an object can't be bound to many variables simultaneously. We could easily have a situation such as the following. When you inspect `d`, you'll see that it contains three items, the numbers 1 and 2, and 3. Multiple assignments, as shown in the example, are legal but bad style.

```
a := b := c := d := OrderedCollection new.
a add: 1.
b add: 2.
c add: 3.
d inspect.
```

In this example, we assign the same instance of OrderedCollection to all the variables. When we modify the collection via one variable, its contents are changed and that change is visible to all variables that have been bound to it. This is illustrated in Figure 4-1.

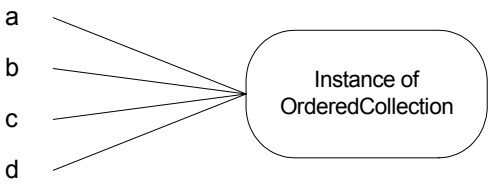


Figure 4-1.
All variables holding same object.

On the other hand, if we were to do the following, we would find that `b` contains the number 2, while `d` contains the numbers 1 and 3.

```
a := b := c := d := OrderedCollection new.
a add: 1.
b := 2.
c add: 3.
b inspect.
d inspect.
```

In this example, we assign a different value to `b`, and bind it to the object 2 instead of the instance of OrderedCollection. This is illustrated in Figure 4-2.

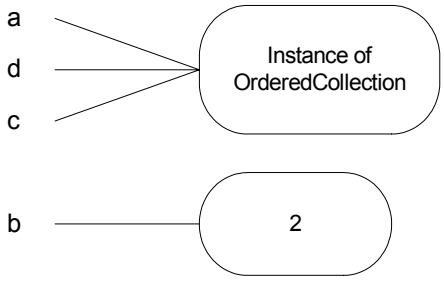


Figure 4-2.
Variables holding different objects.

Accessors

Accessors are methods that allow you to get and set the values of instance variables and class variables. Since instance variables are more heavily used than class variables, you will see accessors most often used with instance variables. Because they get and set variables, accessors are sometimes known as *getters* and *setters*, and when you write accessors, you will usually write both a getter and a setter. The convention is to name them the same as the instance variable. For example,

```
Employee>>salary: aNumber
   salary := aNumber

Employee>>salary
   ^salary
```

If you choose to use lazy initialization (initializing a variable only when it is first needed) rather than to initialize instance variables in an `initialize` method, the `salary` accessor would look something like:

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

Lazy initialization is a reasonable approach when variables are accessed infrequently or not at all, and the cost of initialization is high. Otherwise, it's probably worth initializing variables in an `initialize` method.

Accessors for collections

If your variable contains a collection, what should the getter return? The usual answer is to return a *copy* of the collection. Other classes will be able to take actions based on the contents of the collection, but they won't be able to directly add to or remove from the collection. As well as providing accessors, you'd also provide methods to add to and remove from the collection. For example,

```
Employee>>skills
   ^skillsCollection copy

Employee>>addSkill: aSkill
   ^skillsCollection add: aSkill

Employee>>removeSkill: aSkill
   ^skillsCollection remove: aSkill ifAbsent: [nil]
```

However, you might want to consider not providing *any* direct public interface to your collections. After all, you may decide to change the collection from an *OrderedCollection* to an *Array* or a *Dictionary* and you don't

want to be concerned about how other software is accessing the collection. By providing a few accessing methods such as the `addSkill:` and `removeSkill:` methods shown above, you don't have to expose the actual collection at all. (Another approach would be to provide a getter that always converts the actual collection into an *OrderedCollection* and returns this.)

The five accessor approach

I have a friend who uses what he calls the "fascist" approach to instance variable accessors. For each instance variable, there are five accessors. Using a variable called *salary* as an example, we would have the following accessors in this scheme. The only two public accessors are `salary` and `salary:`. The others are all private. The public accessors are allowed to have side effects (for example, establishing connections), while the private methods are not allowed to have side effects[2]. Not many people use a technique this strict however.

```
Employee>>salary
   "Answer salary. This accessor is allowed to have side effects."
   self myGetSalary == nil
      ifTrue: [self mySetSalary: self myComputeSalary]
   ^self myGetSalary

Employee>>myGetSalary
   "Answer salary. No side effects are allowed."
   ^salary

Employee>>myComputeSalary
   "Answer the initial value for salary, possibly computing it"
   ^self baseSalaryForGrade * self locationFactor

Employee>> salary: aValue
   "Set the salary. This accessor is allowed to have side effects"
   self mySetSalary: aValue.
   self informPayrollSupervisor

Employee>>mySetSalary: aValue
   "Set the salary. No side effects are allowed"
   salary:= aValue
```

Accessors or Direct Referencing

There are schools of thought about how instances should reference their instance variables. One school says that instance variables should *always* be referenced indirectly, through accessor methods. The other school says that *sometimes* instance variables should be referenced *directly*. Let's look at these two ideas.

---

[2] Jumping well ahead of ourselves, subclasses of *ValueModel* provide two setters, **value:** and **setValue:**. This allows objects to use **value:** when they want to notify dependents of changes, and **setValue:** when they want to make a change without notifying dependents. This latter option helps avoid infinite loops when updating a value as a result of a change notification from a dependent. Don't worry if this doesn't make any sense now—we'll cover dependencies later. Below are examples of both methods.

```
ValueModel>>value: newValue
   "Set the currently stored value, and notify dependents."
   self setValue: newValue.
   self changed: #value

ValueHolder>>setValue: aValue
   "Just initialize the value without notifying dependents of a change. "
value := aValue
```

Direct referencing

The advantages of referencing instance variables directly are threefold. First, you don't have to write accessor methods (although the VisualWorks CodingAssistant helps you with this). Second, it's a little more efficient to access them directly because you save a message send on each access (although not much, since the compiler can optimize these message sends). The most significant advantage of referencing variables directly, however, is that you preserve the encapsulation of the object. Any time you write an accessor, the whole world has access to the instance variables, which violates the principle of encapsulation. Sometimes you want to give access to instance variables, but much of the time you'd like to keep private the internal details of how the class works. In *Celebrating 25 Years of Smalltalk*, Ward Cunningham says: "But if I could change anything...I'd like to see people stop giving away all their instance variables with accessing methods."

Referencing through accessors

Requiring all access to go through accessors makes it easier to preserve the public interface to an object when the underlying implementation changes. You may get rid of instance variables or combine them in some way, but as long as you can compute the information that was returned by a getter, and as long as you can make use of the information that was given to a setter, you can preserve the same interface to the world. To use a trivial example, suppose we have an Account object which contains a *balance* instance variable. Perhaps the accessors at one time were:

```
Account>>balance: aFloat
   balance := aFloat

Account>> balance
   ^balance
```

We decide that we'd rather store the value internally in cents rather than in dollars, so we modify the accessors to look like:

```
Account>>balance: aFloat
   balance := (aFloat * 100) rounded

Account>> balance
   ^balance / 100
```

Other advantages to accessors include: If you want to change an instance variable, and instance variables are accessed directly, you have to look through the class and all its subclasses to find references to the variable; on the other hand, if you have accessors on all your instance variables, all you have to do is browse senders of the accessor messages. If you have long methods, it can be difficult to tell what are instance variables and what are temporary variables if you don't use accessors. Having set accessors makes it easier to notify dependents of changes, and to mark an object as modified or dirty when it has changed. For example, combining these last two, you might have a setter that looks like:

```
MyClass>>weight: aNumber
   oldWeight := weight.
   weight := aNumber.
   self markDirty.
   self changed: #weight with: oldWeight
```

Which way to go

My view is that when implementations change, it's usually easier to find and change the references to variables than to preserve an obsolete interface. In code I write for myself, I tend to use accessors only when I want to provide public access to the variables — sometimes this means providing a getter but not a setter. At work, I follow whatever standards prevail (which is usually to use accessors for all instance variables) because I believe it's more important to conform to the standard than be inconsistent.

If you choose to use accessors, but are concerned about violating encapsulation, you can use the *my* prefix described in Chapter 3, Methods. This allows you to use accessors and also preserve the encapsulation, assuming that people don't violate the rules about methods starting with *my*. In Chapter 29, Meta-Programming, we will show an extension to *Class* that automatically creates both public and private accessors for all our instance variables when we define a class.

Chains of accessors

One thing to beware of when writing accessors is creating long chains of accessors to get hold of an object in an object in an object in an object. For example, we might have something like:

```
companyName := employee department division company companyName.
```

The problem with this approach is that it violates encapsulation in a big way. Not only do you know the implementation details of the employee, but you also know and rely on the instance variables for the department, the division, and the company. This makes the system much more fragile because you can't make a change to any of the objects without worrying about how it will affect code. Instead, it's better to encapsulate the inner details. Provide accessors on the outer object that will return inner details. In our example, we might write `company` and `companyName` accessors on the employee, so we can now write:

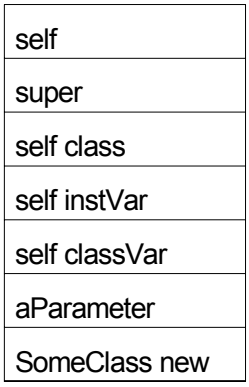
```
companyName := employee companyName.
```

This means that we can restructure how the company is organized, perhaps removing the *division* layer, or perhaps adding a *country* layer, without having to change the code that needs the company name for the employee. Similarly, we would probably write `company` and `companyName` methods on *Department* and on *Division* for the same reason. Our accessors on Employee might look like the following.

```
Employee>>companyName
   ^self company companyName

Employee>>company
   ^self department company
```

This leads to the idea that to maintain loose coupling between classes, a method should send messages only to *self*, *super*, to its class, to instance or class variables, to a parameter, or to an object it creates. If the method sends messages to any other object it is either referencing a global variable, or it knows something about the internal details of the object. This is illustrated in Figure 4-3.




```
someMethod: aParameter
    self foo.
    super foo.
    self class foo.
    self instVarOne foo.
    instVarTwo foo.
    self classVarOne foo.
    classVarTwo foo.
    aParameter foo.
    thing := MyThing new.
    thing foo.
```

Figure 4-2.
Message sends for loose coupling.

Documentation on variables

All instance, class, and class instance variables should be documented in the class comments. You should describe at least the type of the variable, its purpose, and any special information that will be useful to other programmers. For example, here is part of the class comment for Date.

```
Instance Variables:
   day      <Integer>   from 1 to (365 | 366)
   year     <Integer>   typically after the year 1900

Class Variables:
   DaysInMonth    <Array of: Integer>    the number of days in each
month
   MonthNames     <Array of: Symbol>     the names of the 12 months
```

Once you have decided what information your project requires to be in a class comment, you can customize the class comment template by modifying the `commentTemplateString` method in `ClassDescription`.