Pharo is a reflective programming language. In a nutshell, this means that programs are able to reflect on their own execution and structure. More technically, this means that the metaobjects of the runtime system can be reified as ordinary objects, which can be queried and inspected. The metaobjects in Pharo are classes, metaclasses, method dictionaries, compiled methods, but also the run-time stack, processes, and so on. This form of reflection is also called introspection, and is supported by many modern programming languages.

Conversely, it is possible in Pharo to modify reified metaobjects and reflect these changes back to the runtime system (see Figure 1.1). This is also called

---

**Figure 1.1:** Reification and reflection.
intercession, and is supported mainly by dynamic programming languages, and only to a very limited degree by static languages. So pay attention when people say that Java is a reflective language, it is an introspective one not a reflective one.

A program that manipulates other programs (or even itself) is a metaprogram. For a programming language to be reflective, it should support both introspection and intercession. Introspection is the ability to examine the data structures that define the language, such as objects, classes, methods and the execution stack. Intercession is the ability to modify these structures, in other words to change the language semantics and the behavior of a program from within the program itself. Structural reflection is about examining and modifying the structures of the run-time system, and behavioural reflection is about modifying the interpretation of these structures.

In this chapter we will focus mainly on structural reflection. We will explore many practical examples illustrating how Pharo supports introspection and metaprogramming.

1.1 Introspection

Using the inspector, you can look at an object, change the values of its instance variables, and even send messages to it.

Evaluate the following code in a playground:

```
w := GTPlayground openLabel: 'My Playground'.
w inspect
```

This will open a second playground and an inspector. The inspector shows the internal state of this new playground, listing its instance variables on the left (borderColor, borderWidth, bounds...) and the value of the selected instance variable on the right. The bounds instance variable represents the precise area occupied by the playground.

Now choose the inspector and click the playground area of the inspector which has a comment on top and type self bounds: (Rectangle origin: 10@10 corner: 300@300 ) in it of select asshown in Figure 1.2 and then Do It like you do with a code of a Playground.

Immediately you will see the Playground we created change and resize itself.

Accessing instance variables

How does the inspector work? In Pharo, all instance variables are protected. In theory, it is impossible to access them from another object if the class doesn’t define any accessor. In practice, the inspector can access instance variables without needing accessors, because it uses the reflective abilities of Pharo. Classes define instance variables either by name or by numeric
1.1 Introspection

Figure 1.2: Inspecting a Workspace.

indices. The inspector uses methods defined by the `Object` class to access them: `instVarAt:` index and `instVarNamed:` aString can be used to get the value of the instance variable at position index or identified by aString, respectively. Similarly, to assign new values to these instance variables, it uses `instVarAt:put:` and `instVarNamed:put:`.

For instance, you can change the value of the w binding of the first workspace by evaluating:

```
w instVarNamed:'bounds' put: (Rectangle origin: 10@10 corner: 500@500).
```

**Important Caveat:** Although these methods are useful for building development tools, using them to develop conventional applications is a bad idea: these reflective methods break the encapsulation boundary of your objects and can therefore make your code much harder to understand and maintain.

Both `instVarAt:` and `instVarAt:put:` are primitive methods, meaning that they are implemented as primitive operations of the Pharo virtual machine. If you consult the code of these methods, you will see the special pragma syntax `<primitive: N>` where N is an integer.

```
Object >> instVarAt: index
   "Primitive. Answer a fixed variable in an object. ..."

<primitive: 173 error: ec>
   self primitiveFailed
```
Any Pharo code after the primitive declaration is executed only if the primitive fails. This also allows the debugger to be started on primitive methods. In this specific case, there is no way to implement this method, so the whole method just fails.

Other methods are implemented on the VM for faster execution. For example some arithmetic operations on SmallInteger:

```smalltalk
* aNumber
    "Primitive. Multiply the receiver by the argument and answer with the result if it is a SmallInteger. Fail if the argument or the result is not a SmallInteger. Essential. No Lookup. See Object documentation whatIsAPrimitive."

    <primitive: 9>
    ^ super * aNumber
```

If this primitive fails, for example if the VM does not handle the type of the argument, the Pharo code is executed. Although it is possible to modify the code of primitive methods, beware that this can be risky business for the stability of your Pharo system.

Figure 1.3 shows how to display the values of the instance variables of an arbitrary instance (`w`) of class `GTPlayground`. The method `allInstVarNames` returns all the names of the instance variables of a given class.
1.1 Introspection

```plaintext
GTPlayground allInstVarNames
>>> #(#registry #suspendAll #suspendedAnnouncemets #logger #pane #title
 #titleIcon #transformation #actions #condition #implicitNotNil
 #dynamicActionsBlock #color #customValidation #shouldValidate
 #acceptsSelection #parentPrototype #registeredAnnouncers
 #updateActions #selectionActions #selectionDynamicActionsBlock
 #implicitAllNil #rawSelectionTransmissions #statusPane #sourceLink
 #initializationBlock #cachedDisplayedValue #labelActionBlock
 #portChangeActions #wantsSteps #stepTime #stepCondition
 #presentations #arrangement)
```

```plaintext
w := GTPlayground someInstance.
w class allInstVarNames collect: [:each | each -> (w instVarNamed: each)]
```

In the same spirit, it is possible to gather instances that have specific properties iterating over instances of a class using an iterator such as select:. For instance, to get all objects who are directly included in the world morph (the main root of the graphical displayed elements), try this expression:

```plaintext
Morph allSubInstances
  select: [ :each | own |
    own := (each instVarNamed: 'owner').
    own isNotNil and: [ own isWorldMorph ]]
```

### Querying classes and interfaces

The development tools in Pharo (system browser, debugger, inspector...) all use the reflective features we have seen so far.

Here are a few other messages that might be useful to build development tools:

- **isKindOf: aClass** returns true if the receiver is instance of aClass or of one of its superclasses. For instance:

```plaintext
1.5 class
>>> BoxedFloat64
```

```plaintext
1.5 isKindOf: Float
>>> true
```

```plaintext
1.5 isKindOf: Number
>>> true
```

```plaintext
1.5 isKindOf: Integer
>>> false
```

- **respondsTo: aSymbol** returns true if the receiver has a method whose selector is aSymbol. For instance:

```plaintext
```

5
Reflection

```small
1.5 respondsTo: #floor
>>> true "since Number implements floor"

1.5 floor
>>> 1

Exception respondsTo: #,,
>>> true "exception classes can be grouped"
```

**Important Caveat:** Although these features are especially useful for implementing development tools, they are normally not appropriate for typical applications. Asking an object for its class, or querying it to discover which messages it understands, are typical signs of design problems, since they violate the principle of encapsulation. Development tools, however, are not normal applications, since their domain is that of software itself. As such these tools have a right to dig deep into the internal details of code.

**Code metrics**

Let’s see how we can use Pharo’s introspection features to quickly extract some code metrics. Code metrics measure such aspects as the depth of the inheritance hierarchy, the number of direct or indirect subclasses, the number of methods or of instance variables in each class, or the number of locally defined methods or instance variables. Here are a few metrics for the class Morph, which is the superclass of all graphical objects in Pharo, revealing that it is a huge class, and that it is at the root of a huge hierarchy. Maybe it needs some refactoring!

```
"inheritance depth"
Morph allSuperclasses size.
>>> 2

"number of methods"
Morph allSelectors size.
>>> 1304

"number of instance variables"
Morph allInstVarNames size.
>>> 6

"number of new methods"
Morph selectors size.
>>> 896

"number of new variables"
Morph instVarNames size.
>>> 6

"direct subclasses"
Morph subclasses size.
>>> 63
```
One of the most interesting metrics in the domain of object-oriented languages is the number of methods that extend methods inherited from the superclass. This informs us about the relation between the class and its superclasses. In the next sections we will see how to exploit our knowledge of the runtime structure to answer such questions.

### 1.2 Browsing code

In Pharo, everything is an object. In particular, classes are objects that provide useful features for navigating through their instances. Most of the messages we will look at now are implemented in Behavior, so they are understood by all classes.

For example, you can obtain a random instance of a given class by sending it the message *someInstance*.

```plaintext
Point someInstance
>>> 0@0
```

You can also gather all the instances with *allInstances*, or the number of active instances in memory with *instanceCount*.

```plaintext
ByteString allInstances
>>> #('collection' 'position' ...)

ByteString instanceCount
>>> 104565

String allSubInstances size
>>> 101675
```

These features can be very useful when debugging an application, because you can ask a class to enumerate those of its methods exhibiting specific properties. Here are some more interesting and useful methods for code discovery through reflection.

**whichSelectorsAccess**: returns the list of all selectors of methods that read or write the instance variable named by the argument

**whichSelectorsStoreInto**: returns the selectors of methods that modify the value of an instance variable

**whichSelectorsReferTo**: returns the selectors of methods that send a given message
Reflection

Point whichSelectorsAccess: 'x'
>>> #(degrees grid: roundTo: nearestPointAlongLineFrom:to: ...)

Point whichSelectorsStoreInto: 'x'

Point whichSelectorsReferTo: +
>>> an OrderedCollection(#degrees reflectedAbout: grid: ...)

The following messages take inheritance into account:

whichClassIncludesSelector: returns the superclass that implements the given message

unreferencedInstanceVariables returns the list of instance variables that are neither used in the receiver class nor any of its subclasses

Rectangle whichClassIncludesSelector: #inspect
>>> Object

Rectangle unreferencedInstanceVariables
>>> #()

SystemNavigation is a facade that supports various useful methods for querying and browsing the source code of the system. SystemNavigation default returns an instance you can use to navigate the system. For example:

SystemNavigation default allClassesImplementing: #yourself
>>> {Object}

The following messages should also be self-explanatory:

SystemNavigation default allSentMessages size
>>> 370

(SystemNavigation default allUnsentMessagesIn: Object selectors) size
>>> 31

SystemNavigation default allUnimplementedCalls size
>>> 521

Note that messages implemented but not sent are not necessarily useless, since they may be sent implicitly (e.g., using perform:). Messages sent but not implemented, however, are more problematic, because the methods sending these messages will fail at runtime. They may be a sign of unfinished implementation, obsolete APIs, or missing libraries.

Point allCallsOn returns all messages sent explicitly to Point as a receiver.

All these features are integrated into the programming environment of Pharo, in particular the code browsers. As we mentioned before, there are convenient keyboard shortcuts for browsing all implementors (CMD-b CMD-m) and browsing senders (CMD-b CMD-n) of a given message. What is perhaps not so well known is that there are many such pre-packaged queries implemented
1.3 Classes, method dictionaries and methods

as methods of the SystemNavigation class in the browsing protocol. For example, you can programmatically browse all implementors of the message ifTrue: by evaluating:

\[
\text{SystemNavigation default browseAllImplementorsOf: #ifTrue:}
\]

Particularly useful are the methods browseAllSelect: and browseMethodsWithSourceString:matchCase:. Here are two different ways to browse all methods in the system that perform super sends (the first way is rather brute force, the second way is better and eliminates some false positives):

\[
\begin{align*}
\text{SystemNavigation default browseMethodsWithSourceString: 'super' matchCase: true.} \\
\text{SystemNavigation default browseAllSelect: [:method | method sendsToSuper ].}
\end{align*}
\]

1.3 Classes, method dictionaries and methods

Since classes are objects, we can inspect or explore them just like any other object.

Evaluate Point inspect.

In Figure 1.5, the inspector shows the structure of class Point. You can see that the class stores its methods in a dictionary, indexing them by their selector. The selector #* points to the decompiled bytecode of Point>>*. 

Let us consider the relationship between classes and methods. In Figure 1.6 we see that classes and metaclasses have the common superclass Behavior. This is where new is defined, amongst other key methods for classes. Every class has a method dictionary, which maps method selectors to compiled methods.
Each compiled method knows the class in which it is installed. In Figure 1.5 we can even see that this is stored in an association in literal6.

We can exploit the relationships between classes and methods to pose queries about the system. For example, to discover which methods are newly introduced in a given class, i.e., do not override superclass methods, we can navigate from the class to the method dictionary as follows:

```
[:aClass| aClass methodDict keys select: [:aMethod |
    (aClass superclass canUnderstand: aMethod) not ]]
```
A compiled method does not simply store the bytecode of a method. It is also an object that provides numerous useful methods for querying the system. One such method is `isAbstract` (which tells if the method sends subclass-Responsibility). We can use it to identify all the abstract methods of an abstract class.

```smalltalk
[:aClass| aClass methodDict keys select: [[:aMethod | (aClass>>aMethod) isAbstract ]] value: Number

>>> an IdentitySet(#storeOn:base: #printOn:base: #+ #- #* #/ ...)
```

Note that this code sends the `>>` message to a class to obtain the compiled method for a given selector.

To browse the super-sends within a given hierarchy, for example within the Collections hierarchy, we can pose a more sophisticated query:

```smalltalk
class := Collection.
SystemNavigation default
browseMessageList: (class withAllSubclasses gather: [:each |
    each methodDict associations
    select: [:assoc | assoc value sendsToSuper]
thенCollect: [:assoc | RGMethodDefinition realClass: each
        selector: assoc key]])
name: 'Supersends of ', class name, ' and its subclasses'
```

Note how we navigate from classes to method dictionaries to compiled methods to identify the methods we are interested in. A RGMethodDefinition is a lightweight proxy for a compiled method that is used by many tools. There is a convenience method `CompiledMethod>>methodReference` to return the method reference for a compiled method.

```smalltalk
(Object>>#=) methodReference selector

>>> #=
```

### 1.4 Browsing environments

Although `SystemNavigation` offers some useful ways to programmatically query and browse system code, there are more ways. The Browser, which is integrated into Pharo, allows us to restrict the environment in which a search is to perform.

Suppose we are interested to discover which classes refer to the class `Point` but only in its own package.

Open a browser on the class `Point`.

Action-click on the top level package `Kernel` in the package pane and select `Browse scoped`. A new browser opens, showing only the package `Kernel` and all classes within this package (and some classes which have extension methods from this package). Now, in this browser, select again the class `Point`,
Action-click on the class name and select Analyse > Class refs. This will show all methods that have references to the class Point but only those from the package Kernel. Compare this result with the search from a Browser without restricted scope.

This scope is what we call a Browsing Environment (class RBBrowserEnvironment). All other searches, like senders of a method or implementors of a method from within this browser are restricted to this environments too.

Browser environments can also be created programmatically. Here, for example, we create a new RBBrowserEnvironment for Collection and its subclasses, select the super-sending methods, and browse the resulting environment.

```small
((RBBrowserEnvironment new forClasses: (Collection withAllSubclasses))
 selectMethods: [:method | method sendsToSuper])
browse.
```

Note how this is considerably more compact than the earlier, equivalent example using SystemNavigation.

Finally, we can find just those methods that send a different super message programmatically as follows:

```small
((RBBrowserEnvironment new forClasses: (Collection withAllSubclasses))
 selectMethods: [:method |
    method sendsToSuper
    and: [(method parseTree superMessages includes: method selector) not]]
browse
```

Here we ask each compiled method for its (Refactoring Browser) parse tree, in order to find out whether the super messages differ from the method’s selector. Have a look at the querying protocol of the class RBProgramNode to see some the things we can ask of parse trees.

Instead of browsing the environment in a System Browser, we can spawn a MessageBrowser from the list of all methods in this environment.

```small
MessageBrowser browse: ((RBBrowserEnvironment new forClasses:
    (Collection withAllSubclasses))
 selectMethods: [:method |
    method sendsToSuper
    and: [(method parseTree superMessages includes: method selector) not]]) methods
  title: 'Collection methods sending different super'
```

In Figure 1.7 we can see that 5 such methods have been found within the Collection hierarchy, including Collection>>printNameOn:, which sends super printOn:
1.5 Accessing the run-time context

We have seen how Pharo’s reflective capabilities let us query and explore objects, classes and methods. But what about the run-time environment?

Method contexts

In fact, the run-time context of an executing method is in the virtual machine — it is not in the image at all! On the other hand, the debugger obviously has access to this information, and we can happily explore the run-time context, just like any other object. How is this possible?

Actually, there is nothing magical about the debugger. The secret is the pseudo-variable thisContext, which we have encountered only in passing before. Whenever thisContext is referred to in a running method, the entire run-time context of that method is reified and made available to the image as a series of chained Context objects.

We can easily experiment with this mechanism ourselves.

Change the definition of Integer>>factorial by inserting the expression thisContext inspect. self halt. as shown below:

```smalltalk
Integer>>Factorial
"Answer the factorial of the receiver."
self = 0 ifTrue: [thisContext inspect. self halt. ^ 1].
self > 0 ifTrue: [^ self * (self - 1) factorial].
self error: 'Not valid for negative integers'
```
Now evaluate 3 factorial in a workspace. You should obtain both a debugger window and an inspector, as shown in Figure 1.8.

Inspecting thisContext gives you full access to the current execution context, the stack, the local temporaries and arguments, the senders chain and the receiver. Welcome to the poor man’s debugger! If you now browse the class of the explored object (i.e., by evaluating self browse in the bottom pane of the inspector) you will discover that it is an instance of the class Context, as is each sender in the chain.

thisContext is not intended to be used for day-to-day programming, but it is essential for implementing tools like debuggers, and for accessing information about the call stack. You can evaluate the following expression to discover which methods make use of thisContext:

```smalltalk
SystemNavigation default browseMethodsWithSourceString: 'thisContext'
matchCase: true
```

As it turns out, one of the most common applications is to discover the sender of a message. Here is a typical application:

```smalltalk
subclassResponsibility
  "This message sets up a framework for the behavior of the class'
```
subclasses.
Announce that the subclass should have implemented this message."

SubclassResponsibility signalFor: thisContext sender selector

By convention, methods that send self subclassResponsibility are considered to be abstract. But how does Object>>subclassResponsibility provide a useful error message indicating which abstract method has been invoked? Very simply, by asking thisContext for the sender.

**Intelligent breakpoints**

The Pharo way to set a breakpoint is to evaluate self halt at an interesting point in a method. This will cause thisContext to be reified, and a debugger window will open at the breakpoint. Unfortunately this poses problems for methods that are intensively used in the system.

Suppose, for instance, that we want to explore the execution of Morph>>openInWorld. Setting a breakpoint in this method is problematic.

Pay attention the following experiment will break everything! Take a fresh image and set the following breakpoint:

```smalltalk
Morph >> openInWorld
    "Add this morph to the world."
    self halt.
    self openInWorld: self currentWorld
```

Notice how your image immediately freezes as soon as you try to open any new Morph (Menu/Window/...)! We do not even get a debugger window. The problem is clear once we understand that 1) Morph>>openInWorld is used by many parts of the system, so the breakpoint is triggered very soon after we interact with the user interface, but 2) *the debugger itself* sends openInWorld as soon as it opens a window, preventing the debugger from opening! What we need is a way to *conditionally halt* only if we are in a context of interest. This is exactly what Object>>haltIf: offers.

Suppose now that we only want to halt if openInWorld is sent from, say, the context of MorphTest>>testOpenInWorld.

Fire up a fresh image again, and set the following breakpoint:

```smalltalk
Morph>>openInWorld
    "Add this morph to the world."
    self haltIf: #testOpenInWorld.
    self openInWorld: self currentWorld
```

This time the image does not freeze. Try running the MorphTest.

```smalltalk
MorphTest run:#testOpenInWorld.
```
Reflection

How does this work? Let’s have a look at Object>>haltIf:. It first calls if: with the condition to the Exception class Halt. This method itself will check if the condition is a symbol, which is true in this case and finally calls

```smalltalk
Object >> haltIfCallChainContains: aSelector

| ctxt |
ctxt := thisContext.
[ctxt sender isNil] whileFalse: [
  ctxt := ctxt sender.
  (ctxt selector = aSelector) ifTrue: [self signal]].
```

Starting from thisContext, haltIfCallChainContains: goes up through the execution stack, checking if the name of the calling method is the same as the one passed as parameter. If this is the case, then it signals itself, the exception which, by default, summons the debugger.

It is also possible to supply a boolean or a boolean block as an argument to haltIf:, but these cases are straightforward and do not make use of thisContext.

1.6 Intercepting messages not understood

So far we have used Pharo’s reflective features mainly to query and explore objects, classes, methods and the run-time stack. Now we will look at how to use our knowledge of its system structure to intercept messages and modify behaviour at run time.

When an object receives a message, it first looks in the method dictionary of its class for a corresponding method to respond to the message. If no such method exists, it will continue looking up the class hierarchy, until it reaches Object. If still no method is found for that message, the object will send itself the message doesNotUnderstand: with the message selector as its argument. The process then starts all over again, until Object>>doesNotUnderstand: is found, and the debugger is launched.

But what if doesNotUnderstand: is overridden by one of the subclasses of Object in the lookup path? As it turns out, this is a convenient way of realizing certain kinds of very dynamic behaviour. An object that does not understand a message can, by overriding doesNotUnderstand:, fall back to an alternative strategy for responding to that message.

Two very common applications of this technique are 1) to implement lightweight proxies for objects, and 2) to dynamically compile or load missing code.

Lightweight proxies

In the first case, we introduce a minimal object to act as a proxy for an existing object. Since the proxy will implement virtually no methods of its own, any
message sent to it will be trapped by doesNotUnderstand:. By implementing this message, the proxy can then take special action before delegating the message to the real subject it is the proxy for.

Let us have a look at how this may be implemented.

We define a LoggingProxy as follows:

```
ProtoObject subclass: #LoggingProxy
    instanceVariableNames: 'subject invocationCount'
    classVariableNames: ''
    package: 'PBE-Reflection'
```

Note that we subclass ProtoObject rather than Object because we do not want our proxy to inherit around 400 methods (!) from Object.

```
Object methodDict size
>> > 397
```

Our proxy has two instance variables: the subject it is a proxy for, and a count of the number of messages it has intercepted. We initialize the two instance variables and we provide an accessor for the message count. Initially the subject variable points to the proxy object itself.

```
LoggingProxy >> initialize
    invocationCount := 0.
    subject := self.

LoggingProxy >> invocationCount
    ^ invocationCount
```

We simply intercept all messages not understood, print them to the Transcript, update the message count, and forward the message to the real subject.

```
LoggingProxy >> doesNotUnderstand: aMessage
    Transcript show: 'performing ', aMessage printString; cr.
    invocationCount := invocationCount + 1.
    ^ aMessage sendTo: subject
```

Here comes a bit of magic. We create a new Point object and a new LoggingProxy object, and then we tell the proxy to become: the point object:

```
point := 1@2.
LoggingProxy new become: point.
```

This has the effect of swapping all references in the image to the point to now refer to the proxy, and vice versa. Most importantly, the proxy’s subject instance variable will now refer to the point!

```
point invocationCount
>> > 0
point + (3@4)
>> > 4@6
```
This works nicely in most cases, but there are some shortcomings:

Actually the method class is implemented in ProtoObject, but even if it were implemented in Object, which LoggingProxy does not inherit from, it isn’t actually send to the LoggingProxy or its subject. The message is directly answered by the virtual machine. yourself is also never truly sent.

Other messages that may be directly interpreted by the VM, depending on the receiver, include:

Selectors that are never sent, because they are inlined by the compiler and transformed to comparison and jump bytecodes:

Attempts to send these messages to non-boolean normally results in an exception from the VM as it can not use the inlined dispatching for non-boolean receivers. You can intercept this and define the proper behavior by overriding mustBeBoolean in the receiver or by catching the NonBooleanReceiver exception.

Even if we can ignore such special message sends, there is another fundamental problem which cannot be overcome by this approach: self-sends cannot be intercepted:

Our proxy has been cheated out of two self-sends in the rect: method:

Although messages can be intercepted by proxies using this technique, one should be aware of the inherent limitations of using a proxy. In Section 1.7 we will see another, more general approach for intercepting messages.
1.6 Intercepting messages not understood

**Generating missing methods**

The other most common application of intercepting not understood messages is to dynamically load or generate the missing methods. Consider a very large library of classes with many methods. Instead of loading the entire library, we could load a stub for each class in the library. The stubs know where to find the source code of all their methods. The stubs simply trap all messages not understood, and dynamically load the missing methods on demand. At some point, this behaviour can be deactivated, and the loaded code can be saved as the minimal necessary subset for the client application.

Let us look at a simple variant of this technique where we have a class that automatically adds accessors for its instance variables on demand:

```smalltalk
DynamicAccessors >> doesNotUnderstand: aMessage
    | messageName |
    messageName := aMessage selector asString.
    (self class instVarNames includes: messageName)
        ifTrue: [
            self class compile: messageName, String cr, ' ^ ', messageName.
            ^ aMessage sendTo: self ].
    ^ super doesNotUnderstand: aMessage
```

Any message not understood is trapped here. If an instance variable with the same name as the message sent exists, then we ask our class to compile an accessor for that instance variables and we re-send the message.

Suppose the class `DynamicAccessors` has an (uninitialized) instance variable `x` but no pre-defined accessor. Then the following will generate the accessor dynamically and retrieve the value:

```smalltalk
myDA := DynamicAccessors new.
myDA x
>>> nil
```

Let us step through what happens the first time the message `x` is sent to our object (see Figure 1.9).

1. We send `x` to `myDA`, (2) the message is looked up in the class, and (3) not found in the class hierarchy. (4) This causes `self doesNotUnderstand: #x` to be sent back to the object, (5) triggering a new lookup. This time `doesNotUnderstand: #x` is found immediately in `DynamicAccessors`, (6) which asks its class to compile the string `x ^ x`. The compile method is looked up (7), and (8) finally found in `Behavior`, which (9-10) adds the new compiled method to the method dictionary of `DynamicAccessors`. Finally, (11-13) the message is resent, and this time it is found.

The same technique can be used to generate setters for instance variables, or other kinds of boilerplate code, such as visiting methods for a Visitor.

Note the use of `Object>>perform:` in step (12) which can be used to send messages that are composed at run-time:
5 perform: #factorial
   >>> 120

6 perform: ('fac', 'torial') asSymbol
   >>> 720

4 perform: #max: withArguments: (Array with: 6)
   >>> 6

1.7 **Objects as method wrappers**

We have already seen that compiled methods are ordinary objects in Pharo, and they support a number of methods that allow the programmer to query the runtime system. What is perhaps a bit more surprising, is that *any object* can play the role of a compiled method. All it has to do is respond to the method `run:with:in:` and a few other important messages.

Define an empty class Demo. Evaluate `Demo new answer42` and notice how the usual *Message Not Understood* error is raised.

Now we will install a plain object in the method dictionary of our Demo class.

Evaluate `Demo methodDict at: #answer42 put: ObjectsAsMethodsExample new`.

Now try again to print the result of `Demo new answer42`. This time we get the answer 42.

If we take look at the class `ObjectsAsMethodsExample` we will find the following methods:
1.7 Objects as method wrappers

```
answer42

^42

run: oldSelector with: arguments in: aReceiver
^self perform: oldSelector withArguments: arguments
```

When our Demo instance receives the message `answer42`, method lookup proceeds as usual, however the virtual machine will detect that in place of a compiled method, an ordinary Pharo object is trying to play this role. The VM will then send this object a new message `run:with:in:` with the original method selector, arguments and receiver as arguments. Since `ObjectsAsMethodsExample` implements this method, it intercepts the message and delegates it to itself.

We can now remove the fake method as follows:

```
Demo methodDict removeKey: #answer42 ifAbsent: []
```

If we take a closer look at `ObjectsAsMethodsExample`, we will see that its superclass also implements some methods like `flushcache`, `methodClass:` and `selector:`, but they are all empty. These messages may be sent to a compiled method, so they need to be implemented by an object pretending to be a compiled method. (`flushcache` is the most important method to be implemented; others may be required by some tools and depending on whether the method is installed using `Behavior>>addSelector:withMethod:or` directly using `MethodDictionary>>at:put:`.)

Using method wrappers to perform test coverage

Method wrappers are a well-known technique for intercepting messages. In the original implementation (http://www.squeaksource.com/MethodWrappers.html), a method wrapper is an instance of a subclass of `CompiledMethod`. When installed, a method wrapper can perform special actions before or after invoking the original method. When uninstalled, the original method is returned to its rightful position in the method dictionary.

In Pharo, method wrappers can be implemented more easily by implementing `run:with:in:` instead of by subclassing `CompiledMethod`. In fact, there exists a lightweight implementation of objects as method wrappers (http://www.squeaksource.com/ObjectsAsMethodsWrap.html), but it is not part of standard Pharo at the time of this writing.

Nevertheless, the Pharo Test Runner uses precisely this technique to evaluate test coverage. Let's have a quick look at how it works.

The entry point for test coverage is the method `TestRunner>>runCoverage`:

```
TestRunner >> runCoverage

| packages methods |

... "identify methods to check for coverage"

self collectCoverageFor: methods
```
The method `TestRunner>>collectCoverageFor:` clearly illustrates the coverage checking algorithm:

```smalltalk
TestRunner >> collectCoverageFor: methods
    | wrappers suite |
    wrappers := methods collect: [:each | TestCoverage on: each ].
    suite := self
        resetResult;
        suiteForAllSelected.

    [ wrappers do: [:each | each install ].

    wrappers := wrappers reject: [:each | each hasRun].
    wrappers := wrappers collect: [:each | each reference].
    wrappers isEmpty
        ifTrue: [ UIManager default inform: 'Congratulations. Your tests cover all code under analysis.' ]
        ifFalse: ...
```

A wrapper is created for each method to be checked, and each wrapper is installed. The tests are run, and all wrappers are uninstalled. Finally the user obtains feedback concerning the methods that have not been covered.

How does the wrapper itself work? The `TestCoverage` wrapper has three instance variables, `hasRun`, `reference` and `method`. They are initialized as follows:

```smalltalk
TestCoverage class >> on: aMethodReference
    ^ self new initializeOn: aMethodReference

TestCoverage >> initializeOn: aMethodReference
    hasRun := false.
    method := reference compiledMethod
```

The install and uninstall methods simply update the method dictionary in the obvious way:

```smalltalk
TestCoverage >> install
    reference actualClass methodDict
        at: reference selector
        put: self

TestCoverage >> uninstall
    reference actualClass methodDict
        at: reference selector
        put: method
```

The `run:with:in:` method simply updates the `hasRun` variable, uninstalls the wrapper (since coverage has been verified), and resends the message to the
original method.

```smalltalk
run: aSelector with: anArray in: aReceiver
    self mark; uninstall.
    ^ aReceiver withArgs: anArray executeMethod: method
mark
    hasRun := true
```

Take a look at `ProtoObject>>withArgs:executeMethod:` to see how a method displaced from its method dictionary can be invoked.

That’s all there is to it!

Method wrappers can be used to perform any kind of suitable behaviour before or after the normal operation of a method. Typical applications are instrumentation (collecting statistics about the calling patterns of methods), checking optional pre- and post-conditions, and memoization (optionally caching computed values of methods).

## 1.8 Pragmas

A **pragma** is an annotation that specifies data about a program, but is not involved in the execution of the program. Pragmas have no direct effect on the operation of the method they annotate. Pragmas have a number of uses, among them:

* **Information for the compiler:** pragmas can be used by the compiler to make a method call a primitive function. This function has to be defined by the virtual machine or by an external plug-in.

* **Runtime processing:** Some pragmas are available to be examined at runtime.

Pragmas can be applied to a program’s method declarations only. A method may declare one or more pragmas, and the pragmas have to be declared prior any Smalltalk statement. Each pragma is in effect a static message send with literal arguments.

We briefly saw pragmas when we introduced primitives earlier in this chapter. A primitive is nothing more than a pragma declaration. Consider `<primitive: 173 error:ec>` as contained in `instVarAt:`. The pragma’s selector is `primitive:error:` and its arguments is an immediate literal value, 173. The variable `ec` is an error code, filled by the VM in case the execution of the implementation on the VM side failed.

The compiler is probably the bigger user of pragmas. SUnit is another tool that makes use of annotations. SUnit is able to estimate the coverage of an application from a test unit. One may want to exclude some methods from the coverage. This is the case of the documentation method in `SplitJointTest` class:
Reflection

SplitJointTest class >> documentation
<ignoreForCoverage>
"self showDocumentation"

^ 'This package provides function.... "

By simply annotating a method with the pragma <ignoreForCoverage> one can control the scope of the coverage.

As instances of the class Pragma, pragmas are first class objects. A compiled method answers to the message pragmas. This method returns an array of pragmas.

(SplitJointTest class >> #showDocumentation) pragmas.
>>> an Array(<ignoreForCoverage>)
(Float>>#+) pragmas
>>> an Array(<primitive: 41>)

Methods defining a particular query may be retrieved from a class. The class side of SplitJointTest contains some methods annotated with <ignoreForCoverage>:

Pragma allNamed: #ignoreForCoverage in: SplitJointTest class
>>> an Array(<ignoreForCoverage> <ignoreForCoverage> <ignoreForCoverage>)

A variant of allNamed:in: may be found on the class side of Pragma.

A pragma knows in which method it is defined (using method), the name of the method (selector), the class that contains the method (methodClass), its number of arguments (numArgs), about the literals the pragma has for arguments (hasLiteral: and hasLiteralSuchThat:).

1.9 Chapter summary

Reflection refers to the ability to query, examine and even modify the metaobjects of the runtime system as ordinary objects.

• The Inspector uses instVarAt: and related methods to view private instance variables of objects.

• Send Behavior>>allInstances to query instances of a class.

• The messages class, isKindOf:, respondsTo: etc. are useful for gathering metrics or building development tools, but they should be avoided in regular applications: they violate the encapsulation of objects and make your code harder to understand and maintain.

• SystemNavigation is a utility class holding many useful queries for navigation and browsing the class hierarchy. For example, use SystemNavigation default browseMethodsWithSourceString: 'pharo'
matchCase: true to find and browse all methods with a given source string. (Slow, but thorough!)

- Every Pharo class points to an instance of MethodDictionary which maps selectors to instances of CompiledMethod. A compiled method knows its class, closing the loop.

- RGMethodDefinition is a lightweight proxy for a compiled method, providing additional convenience methods, and used by many Pharo tools.

- RBBrowserEnvironment, part of the Refactoring Browser infrastructure, offers a more refined interface than SystemNavigation for querying the system, since the result of a query can be used as the scope of a new query. Both GUI and programmatic interfaces are available.

- thisContext is a pseudo-variable that reifies the runtime stack of the virtual machine. It is mainly used by the debugger to dynamically construct an interactive view of the stack. It is also especially useful for dynamically determining the sender of a message.

- Intelligent breakpoints can be set using haltIf:, taking a method selector as its argument. haltIf: halts only if the named method occurs as a sender in the run-time stack.

- A common way to intercept messages sent to a given target is to use a minimal object as a proxy for that target. The proxy implements as few methods as possible, and traps all message sends by implementing doesNotUnderstand:. It can then perform some additional action and then forward the message to the original target.

- Send become: to swap the references of two objects, such as a proxy and its target.

- Beware, some messages, like class and yourself are never really sent, but are interpreted by the VM. Others, like +, - and ifTrue: may be directly interpreted or inlined by the VM depending on the receiver.

- Another typical use for overriding doesNotUnderstand: is to lazily load or compile missing methods.

- doesNotUnderstand: cannot trap self-sends.

- A more rigorous way to intercept messages is to use an object as a method wrapper. Such an object is installed in a method dictionary in place of a compiled method. It should implement run:with:in: which is sent by the VM when it detects an ordinary object instead of a compiled method in the method dictionary. This technique is used by the SUnit Test Runner to collect coverage data.