The Pharo object model

The Pharo programming model is heavily inspired by the one of Smalltalk. It is simple and uniform: everything is an object, and objects communicate only by sending each other messages. Instance variables are private to the object. Methods are all public and dynamically looked up (late-bound). In this chapter we present the core concepts of the Pharo object model. We revisit concepts such as self and super and define precisely their semantics. Then we discuss the consequences of representing classes as objects. This will be extended in Chapter: Classes and Metaclasses.

1.1 The rules of the model

The object model is based on a set of simple rules that are applied uniformly. The rules are as follows:

Rule 1. Everything is an object.

Rule 2. Every object is an instance of a class.

Rule 3. Every class has a superclass.

Rule 4. Everything happens by sending messages.

Rule 5. Method lookup dynamically follows the inheritance chain.

Let us look at each of these rules in some detail.

1.2 Everything is an Object

The mantra *everything is an object* is highly contagious. After only a short while working with Pharo, you will start to be surprised at how this rule
simplifies everything you do. Integers, for example, are truly objects, so you can send messages to them, just as you do to any other object. At the end of this chapter, we added an implementation note on the object implementation for the curious reader.

"send '+ 4' to 3, yielding 7"
3 + 4
>>> 7

"send factorial, yielding a big number"
20 factorial
>>> 2432902008176640000

The object 7 is different than the object returned by 20 factorial, but because they are both polymorphic objects, none of the code, not even the implementation of factorial, needs to know about this.

Coming back to everything is an object rule, perhaps the most fundamental consequence of this rule is that classes are objects too. Classes are not second-class objects: they are really first-class objects that you can send messages to, inspect, and change. This means that Pharo is a truly reflective system, which gives a great deal of expressive power to developers.

Important Classes are objects too.

1.3 Every object is an instance of a class

Every object has a class; you can find out which one by sending it the message class.

1 class
>>> SmallInteger

20 factorial class
>>> LargePositiveInteger

'h hello' class
>>> ByteString

(4@5) class
>>> Point

Object new class
>>> Object

A class defines the structure of its instances via instance variables, and the behavior of its instances via methods. Each method has a name, called its selector, which is unique within the class.

Since classes are objects, and every object is an instance of a class, it follows that classes must also be instances of classes. A class whose instances are classes
is called a *metaclass*. Whenever you create a class, the system automatically creates a metaclass for you. The metaclass defines the structure and behavior of the class that is its instance. 99% of the time you will not need to think about metaclasses, and may happily ignore them. (We will have a closer look at metaclasses in Chapter : Classes and Metaclasses.)

1.4 **Instance structure and behavior**

Now we will briefly present how we specify the structure and behavior of instances.

**Instance variables**

Instance variables in Pharo are private to the *instance* itself. This is in contrast to Java and C++, which allow instance variables (also known as *fields* or *member variables*) to be accessed by any other instance that happens to be of the same class. We say that the *encapsulation boundary* of objects in Java and C++ is the class, whereas in Pharo it is the instance.

In Pharo, two instances of the same class cannot access each other’s instance variables unless the class defines *accessor methods*. There is no language syntax that provides direct access to the instance variables of any other object. (Actually, a mechanism called reflection does provide a way to ask another object for the values of its instance variables; meta-programming is intended for writing tools like the object inspector, whose sole purpose is to look inside other objects.)

Instance variables can be accessed by name in any of the instance methods of the class that defines them, and also in the methods defined in its subclasses. This means that Pharo instance variables are similar to *protected* variables in C++ and Java. However, we prefer to say that they are private, because it is considered bad style in Pharo to access an instance variable directly from a subclass.

**Instance encapsulation example**

The method `Point>>dist:` computes the distance between the receiver and another point. The instance variables `x` and `y` of the receiver are accessed directly by the method body. However, the instance variables of the other point must be accessed by sending it the messages `x` and `y`.

```plaintext
Point >> dist: aPoint
  "Answer the distance between aPoint and the receiver."
  | dx dy |
  dx := aPoint x - x.
  dy := aPoint y - y.
```
The Pharo object model

\[
\sqrt{(dx \times dx) + (dy \times dy)}
\]

1 \@ 1  dist: 4 \@ 5

\[
>>> 5.0
\]

The key reason to prefer instance-based encapsulation to class-based encapsulation is that it enables different implementations of the same abstraction to coexist. For example, the method `dist:` need not know or care whether the argument `aPoint` is an instance of the same class as the receiver. The argument object might be represented in polar coordinates, or as a record in a database, or on another computer in a distributed system. As long as it can respond to the messages `x` and `y`, the code of method `dist:` (shown above) will still work.

**Methods**

All methods are public and virtual (i.e., dynamically looked up). Methods are grouped into protocols that indicate their intent. Some common protocol names have been established by convention, for example, accessing for all accessor methods, and initialization for establishing a consistent initial state for the object. The protocol private is sometimes used to group methods that should not be seen from outside. Nothing, however, prevents you from sending a message that is implemented by such a "private" method.

Methods can access all instance variables of the object. Some developers prefer to access instance variables only through accessors. This practice has some value, but it also clutters the interface of your classes, and worse, exposes its private state to the world.

**1.5 The instance side and the class side**

Since classes are objects, they can have their own instance variables and their own methods. We call these class instance variables and class methods, but they are really no different from ordinary instance variables and methods: They simply operate on different objects (classes in this case). An instance variable describes instance state and a method describes instance behavior. Similarly, class instance variables are just instance variables defined by a metaclass (and describe the state of classes - instances of metaclasses), and class methods are just methods defined by a metaclass (and that will be executed on classes).

A class and its metaclass are two separate classes, even though the former is an instance of the latter. However, this is largely irrelevant to you as a programmer: you are concerned with defining the behavior of your objects and the classes that create them.

For this reason, the browser helps you to browse both class and metaclass as if they were a single thing with two "sides": the instance side and the class side, as shown in Figure 1.1. By default, when you select a class in the browser, you're
browsing the instance side (i.e., the methods that are executed when messages are sent to an instance of Color). Clicking on the Class side button switches you over to the class side (the methods that will be executed when messages are sent to the class Color itself).

For example, Color blue sends the message blue to the class Color. You will therefore find the method blue defined on the class side of Color, not on the instance side.

```
"Class-side method blue (convenient instance creation method)"
aColor := Color blue.
>>> Color blue
"Color instances are self-evaluating"

"Instance-side accessor method red (returns the red RGB value)"
Color blue red
>>> 0.0

"Instance-side accessor method blue (returns the blue RGB value)"
Color blue blue
>>> 1.0
```

You define a class by filling in the template proposed on the instance side. When you accept this template, the system creates not just the class that you defined, but also the corresponding metaclass (which you can then edit by clicking on the Class side button). The only part of the metaclass creation template that makes sense for you to edit directly is the list of the metaclass’s instance variable names.

Once a class has been created, browsing its instance side (Class side unchecked) lets you edit and browse the methods that will be possessed by instances of that class (and of its subclasses).
Class methods

Class methods can be quite useful; browse Color class for some good examples. You will see that there are two kinds of methods defined on a class: instance creation methods, like Color class>>blue, and those that perform a utility function, like Color class>>wheel:. This is typical, although you will occasionally find class methods used in other ways.

It is convenient to place utility methods on the class side because they can be executed without having to create any additional objects first. Indeed, many of them will contain a comment designed to make it easy to execute them.

Browse method Color class>>wheel:, double-click just at the beginning of the comment "(Color wheel: 12) inspect" and press CMD-d. You will see the effect of executing this method.

For those familiar with Java and C++, class methods may seem similar to static methods. However, the uniformity of the Pharo object model (where classes are just regular objects) means that they are somewhat different: whereas Java static methods are really just statically-resolved procedures, Pharo class methods are dynamically-dispatched methods. This means that inheritance, overriding and super-sends work for class methods in Pharo, whereas they don’t work for static methods in Java.

Class instance variables

With ordinary instance variables, all the instances of a class have the same set of variable names (though each instance has its own private set of values), and the instances of its subclasses inherit those names. The story is exactly the same with class instance variables: each class has its own private class instance variables. A subclass will inherit those class instance variables, but the subclass will have its own private copies of those variables. Just as objects don’t share instance variables, neither do classes and their subclasses share class instance variables.

For example, you could use a class instance variable called count to keep track of how many instances you create of a given class. However, any subclass would have its own count variable, so subclass instances would be counted separately.

Example: Class instance variables and subclasses

Suppose we define the class Dog, and its subclass Hyena. Suppose that we add a count class instance variable to the class Dog (i.e. we define it on the metaclass Dog class). Hyena will naturally inherit the class instance variable count from Dog.

```plaintext
Object subclass: #Dog
    instanceVariableNames: ''
```
1.5 The instance side and the class side

```plaintext
Dog class
  instanceVariableNames: 'count'

Hyena subclass: #Hyena
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'PBE-CIV'
```

Now suppose we define class methods for `Dog` to initialize its `count` to 0, and to increment it when new instances are created:

```plaintext
Dog class >> initialize
  count := 0.

Dog class >> new
  count := count + 1.
  ^ super new

Dog class >> count
  ^ count
```

Now when we create a new `Dog`, the `count` value of the class `Dog` is incremented, and so is that of the class `Hyena` (but the hyenas are counted separately).

*Side note:* Notice the use of `initialize` on the classes, in the following code. In Pharo, when you instantiate an object such as `Dog new`, `initialize` is called automatically as part of the `new` message send (you can see for yourself by browsing `Behavior>>new`). But with classes, simply defining them does not automatically call `initialize`, and so we have to call it explicitly here. By default class `initialize` methods are automatically executed only when classes are loaded. See also the discussion about lazy initialization, below.

```plaintext
Dog initialize.
Hyena initialize.
Dog count
  >>> 0

Hyena count
  >>> 0
```

```plaintext
<table>
<thead>
<tr>
<th>aDog</th>
</tr>
</thead>
</table>
aDog := Dog new.
Dog count
  >>> 1 "Incremented"

Hyena count
  >>> 0 "Still the same"
```

Class instance variables are private to a class in exactly the same way that instance variables are private to an instance. Since classes and their instances are different objects, this has the following consequences:
1. A class does not have access to the instance variables of its own instances. So, the class Color does not have access to the variables of an object instantiated from it, aColorRed. In other words, just because a class was used to create an instance (using new or a helper instance creation method like Color red), it doesn’t give the class any special direct access to that instance’s variables. The class instead has to go through the accessor methods (a public interface) just like any other object.

2. The reverse is also true: an instance of a class does not have access to the class instance variables of its class. In our example above, aDog (an individual instance) does not have direct access to the count variable of the Dog class (except, again, through an accessor method).

**Important** A class does not have access to the instance variables of its own instances. An instance of a class does not have access to the class instance variables of its class.

For this reason, instance initialization methods must always be defined on the instance side, the class side has no access to instance variables, and so cannot initialize them! All that the class can do is to send initialization messages, using accessors, to newly created instances.

Java has nothing equivalent to class instance variables. Java and C++ static variables are more like Pharo class variables (discussed in Section 1.9), since in those languages all of the subclasses and all of their instances share the same static variable.

**Example: Defining a Singleton**

The Singleton pattern provides a typical example of the use of class instance variables and class methods. Imagine that we would like to implement a class WebServer, and to use the Singleton pattern to ensure that it has only one instance.

We define the class WebServer as follow.

```object
Object subclass: #WebServer
  instanceVariableNames: 'sessions'
  classVariableNames: ''
  package: 'Web'
```

Then, clicking on the Class side button, we add the (class) instance variable uniqueInstance.

```webserver
WebServer class
  instanceVariableNames: 'uniqueInstance'
```

As a result, the class WebServer class will have a new instance variable (in addition to the variables that it inherits from Behavior, such as superclass
and methodDict). It means that the value of this extra instance variable will describe the instance of the class WebServer class i.e. the class WebServer.

Object class allInstVarNames
>>> "#('superclass' 'methodDict' 'format' 'layout' 'instanceVariables'
  'organization' 'subclasses' 'name' 'classPool' 'sharedPools'
  'environment' 'category' 'traitComposition' 'localSelectors')"

WebServer class allInstVarNames
>>>"#('superclass' 'methodDict' 'format' 'layout' 'instanceVariables'
  'organization' 'subclasses' 'name' 'classPool' 'sharedPools'
  'environment' 'category' 'traitComposition' 'localSelectors'
  #uniqueInstance)"

We can now define a class method named uniqueInstance, as shown below. This method first checks whether uniqueInstance has been initialized. If it has not, the method creates an instance and assigns it to the class instance variable uniqueInstance. Finally the value of uniqueInstance is returned. Since uniqueInstance is a class instance variable, this method can directly access it.

WebServer class >> uniqueInstance
  uniqueInstance ifNil: [ uniqueInstance := self new ].
  ^ uniqueInstance

The first time that WebServer uniqueInstance is executed, an instance of the class WebServer will be created and assigned to the uniqueInstance variable. The next time, the previously created instance will be returned instead of creating a new one. (This pattern, checking if a variable is nil in an accessor method, and initializing its value if it is nil, is called lazy initialization).

Note that the instance creation code in the code above.

is written as self new and not as WebServer new. What is the difference? Since the uniqueInstance method is defined in WebServer class, you might think that there is no difference. And indeed, until someone creates a subclass of WebServer, they are the same. But suppose that ReliableWebServer is a subclass of WebServer, and inherits the uniqueInstance method. We would clearly expect ReliableWebServer uniqueInstance to answer a ReliableWebServer. Using self ensures that this will happen, since self will be bound to the respective receiver, here the classes WebServer and ReliableWebServer. Note also that WebServer and ReliableWebServer will each have a different value for their uniqueInstance instance variable.

A note on lazy initialization. Do not over-use the lazy initialization pattern. The setting of initial values for instances of objects generally belongs in the initialize method. Putting initialization calls only in initialize helps from a readability perspective – you don’t have to hunt through all the accessor methods to see what the initial values are. Although it may be tempting to instead
initialize instance variables in their respective accessor methods (using ifNil: checks), avoid this unless you have a good reason.

For example, in our uniqueInstance method above, we used lazy initialization because users won’t typically expect to call WebServer initialize. Instead, they expect the class to be “ready” to return new unique instances. Because of this, lazy initialization makes sense. Similarly, if a variable is expensive to initialize (opening a database connection or a network socket, for example), you will sometimes choose to delay that initialization until you actually need it.

1.6 **Every class has a superclass**

Each class in Pharo inherits its behaviour and the description of its structure from a single *superclass*. This means that Smalltalk has single inheritance.

```
SmallInteger superclass
>>> Integer
```

```
Integer superclass
>>> Number
```

```
Number superclass
>>> Magnitude
```

```
Magnitude superclass
>>> Object
```

```
Object superclass
>>> ProtoObject
```

```
ProtoObject superclass
>>> nil
```

Traditionally the root of an inheritance hierarchy is the class Object (since everything is an object). In Pharo, the root is actually a class called ProtoObject, but you will normally not pay any attention to this class. ProtoObject encapsulates the minimal set of messages that all objects *must* have and ProtoObject is designed to raise as many as possible errors (to support proxy definition). However, most classes inherit from Object, which defines many additional messages that almost all objects understand and respond to. Unless you have a very good reason to do otherwise, when creating application classes you should normally subclass Object, or one of its subclasses.

A new class is normally created by sending the message subclass: instance-VariableNames: ... to an existing class. There are a few other methods to create classes. To see what they are, have a look at Class and its subclass creation protocol.

Although Pharo does not provide multiple inheritance, it supports a mechanism called Traits for sharing behaviour across unrelated classes. Traits are
collections of methods that can be reused by multiple classes that are not related by inheritance. Using traits allows one to share code between different classes without duplicating code.

**Abstract methods and abstract classes**

An abstract class is a class that exists to be subclassed, rather than to be instantiated. An abstract class is usually incomplete, in the sense that it does not define all of the methods that it uses. The "placeholder" methods, those that the other methods assume to be (re)defined are called abstract methods.

Pharo has no dedicated syntax to specify that a method or a class is abstract. Instead, by convention, the body of an abstract method consists of the expression `self subclassResponsibility`. This indicates that subclasses have the responsibility to define a concrete version of the method. `self subclassResponsibility` methods should always be overridden, and thus should never be executed. If you forget to override one, and it is executed, an exception will be raised.

Similarly, a class is considered abstract if one of its methods is abstract. Nothing actually prevents you from creating an instance of an abstract class; everything will work until an abstract method is invoked.

**Example: the abstract class Magnitude**

Magnitude is an abstract class that helps us to define objects that can be compared to each other. Subclasses of Magnitude should implement the methods `<`, `=`, and `hash`. Using such messages, Magnitude defines other methods such as `>`, `>=`, `=`; `max`, `min; between:and: ` and others for comparing objects. Such methods are inherited by subclasses. The method `Magnitude>><` is abstract, and defined as shown in the following script.

```smalltalk
< aMagnitude
    "Answer whether the receiver is less than the argument."
    ^self subclassResponsibility
```

By contrast, the method `>=` is concrete, and is defined in terms of `<`.

```smalltalk
>= aMagnitude
    "Answer whether the receiver is greater than or equal to the argument."
    ^(self < aMagnitude) not
```

The same is true of the other comparison methods (they are all defined in terms of the abstract method `<`).
Character is a subclass of Magnitude; it overrides the < method (which, if you recall, is marked as abstract in Magnitude by the use of self subclassResponsibility) with its own version (see the method definition below).

Character also explicitly defines methods = and hash; it inherits from Magnitude the methods >=, <=, ~= and others.

```small
< aCharacter
    "Answer true if the receiver's value < aCharacter's value."
    ^self asciiValue < aCharacter asciiValue
```

**Traits**

A *trait* is a collection of methods that can be included in the behaviour of a class without the need for inheritance. This makes it easy for classes to have a unique superclass, yet still share useful methods with otherwise unrelated classes.

To define a new trait, simply right-click in the class pane and select Add Trait, or replace the subclass creation template by the trait creation template, below.

```plaintext
Trait named: #TAuthor
    uses: { }
    package: 'PBE-LightsOut'
```

Here we define the trait TAuthor in the package PBE-LightsOut. This trait does not use any other existing traits. In general we can specify a trait composition expression of other traits to use as part of the uses: keyword argument. Here we simply provide an empty array.

Traits may contain methods, but no instance variables. Suppose we would like to be able to add an author method to various classes, independent of where they occur in the hierarchy.

We might do this as follows:

```plaintext
TAuthor >> author
    "Returns author initials"
    ^ 'on' "oscar nierstrasz"
```

Now we can use this trait in a class that already has its own superclass, for instance the LOGame class that we defined in Chapter : A First Application. We simply modify the class creation template for LOGame to include a uses: keyword argument that specifies that TAuthor should be used.

```plaintext
BorderedMorph subclass: #LOGame
    uses: TAuthor
    instanceVariableNames: 'cells'
    classVariableNames: ''
```
1.7 Everything happens by sending messages

```plaintext
package: 'PBE-LightsOut'

If we now instantiate LOGame, it will respond to the author message as ex-
pected.

```LOGame new author
```[author]
```[on]

Trait composition expressions may combine multiple traits using the + opera-
tor. In case of conflicts (i.e., if multiple traits define methods with the same
name), these conflicts can be resolved by explicitly removing these meth-
ods with -), or by redefining these methods in the class or trait that you are
defining. It is also possible to alias methods (with @), providing a new name for
them.

Traits are used in the system kernel. One good example is the class Behavior.

Object subclass: #Behavior
uses: TPureBehavior @
  [#basicAddTraitSelector:withMethod:->#addTraitSelector:withMethod:]
instanceVariableNames: 'superclass methodDict format'
classVariableNames: 'ObsoleteSubclasses'
package: 'Kernel-Classes'

Here we see that the method addTraitSelector:withMethod: defined in
the trait TPureBehavior has been aliased to basicAddTraitSelector:with-
Method:

1.7 Everything happens by sending messages

This rule captures the essence of programming in Pharo.

In procedural programming (and in some static features of some object-
oriented languages such as Java), the choice of which piece of code to execute
when a procedure is called is made by the caller. The caller chooses the proce-
dure to execute statically, by name.

In Pharo, we do not "invoke methods". Instead, we send messages. This is
just a terminology point but it is significant. It implies that this is not the
responsibility of the client to select the method to be executed, it is the one of
the receiver of the message.

When sending a message, we do not decide which method will be executed.
Instead, we tell an object to do something for us by sending it a message. A
message is nothing but a name and a list of arguments. The receiver then
decides how to respond by selecting its own method for doing what was asked.
Since different objects may have different methods for responding to the
same message, the method must be chosen dynamically, when the message is
received.
As a consequence, we can send the same message to different objects, each of which may have its own method for responding to the message. We do not tell the SmallInteger 3 or the Point (1@2) how to respond to the message + 4. Each has its own method for +, and responds to + 4 accordingly.

One of the consequences of Pharo’s model of message sending is that it encourages a style in which objects tend to have very small methods and delegate tasks to other objects, rather than implementing huge, procedural methods that assume too much responsibility. Joseph Pelrine expresses this principle succinctly as follows:

"Don’t do anything that you can push off onto someone else."

Many object-oriented languages provide both static and dynamic operations for objects. In Pharo there are only dynamic message sends. For example, instead of providing static class operations, we simply send messages to classes (which are simply objects).

Ok, so nearly everything in Pharo happens by sending messages. At some point action must take place:

Variable declarations are not message sends. In fact, variable declarations are not even executable. Declaring a variable just causes space to be allocated for an object reference.

Assignments are not message sends. An assignment to a variable causes that variable name to be freshly bound in the scope of its definition.

Returns are not message sends. A return simply causes the computed result to be returned to the sender.

Primitives (and Pragmas/annotations) are not message sends. They are implemented in the virtual machine.

Other than these few exceptions, pretty much everything else does truly happen by sending messages. In particular, since there are no public fields in Pharo, the only way to update an instance variable of another object is to send it a message asking that it update its own field. Of course, providing setter and getter methods for all the instance variables of an object is not good object-oriented style. Joseph Pelrine also states this very nicely:

"Don’t let anyone else play with your data."
1.8 Method lookup follows the inheritance chain

What exactly happens when an object receives a message? This is a two step process: method lookup and method execution.

Lookup. First, the method having the same name as the message is looked up.

Method Execution. Second, the found method is applied to the receiver with the message arguments: When the method is found, the arguments are bound to the parameters of the method, and the virtual machine executes it.

The lookup process is quite simple:

1. The class of the receiver looks up the method to use to handle the message.
2. If this class does not have that method method defined, it asks its superclass, and so on, up the inheritance chain.

It is essentially as simple as that. Nevertheless there are a few questions that need some care to answer:

- What happens when a method does not explicitly return a value?
- What happens when a class reimplements a superclass method?
- What is the difference between self and super sends?
- What happens when no method is found?

The rules for method lookup that we present here are conceptual; virtual machine implementors use all kinds of tricks and optimizations to speed up method lookup. That’s their job, but you should never be able to detect that they are doing something different from our rules.

First let us look at the basic lookup strategy, and then consider these further questions.

Method lookup

Suppose we create an instance of EllipseMorph.

```smalltalk
anEllipse := EllipseMorph new.
```

If we now send this object the message defaultColor, we get the result Color yellow.

```smalltalk
anEllipse defaultColor
>>> Color yellow
```

The class EllipseMorph implements defaultColor, so the appropriate method is found immediately.

```smalltalk
EllipseMorph >> defaultColor
    "Answer the default color/fill style for the receiver"
    ^ Color yellow
```
Figure 1.2: Method lookup follows the inheritance hierarchy

In contrast, if we send the message `openInWorld` to `anEllipse`, the method is not immediately found, since the class `EllipseMorph` does not implement `openInWorld`. The search therefore continues in the superclass, `BorderedMorph`, and so on, until an `openInWorld` method is found in the class `Morph` (see Figure 1.2).

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

**Returning self**

Notice that `EllipseMorph>>defaultColor` explicitly returns `Color yellow`, whereas `Morph>>openInWorld` does not appear to return anything.

Actually a method *always* answers a message with a value (which is, of course, an object). The answer may be defined by the `^` construct in the method, but if execution reaches the end of the method without executing a `^`, the method still answers a value – it answers the object that received the message. We usually say that the method answers *self*, because in Pharo the pseudo-variable `self` represents the receiver of the message, much like the keyword `this` in Java. Other languages, such as Ruby, by default return the value of the last statement in the method. Again, this is not the case in Pharo, instead you can imagine that a method without an explicit return ends with `^ self`.

**Important**  `self` represents the receiver of the message.

This suggests that `openInWorld` is equivalent to `openInWorldReturnSelf`, defined below.
1.8 Method lookup follows the inheritance chain

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

Why is explicitly writing `^ self` not a so good thing to do? When you return something explicitly, you are communicating that you are returning something of interest to the sender. When you explicitly return `self`, you are saying that you expect the sender to use the returned value. This is not the case here, so it is best not to explicitly return `self`. We only return `self` on special case to stress that the receiver is returned.

This is a common idiom in Pharo, which Kent Beck refers to as **Interesting return value**:

"Return a value only when you intend for the sender to use the value."

**Important** By default (if not specified differently) a method returns the message receiver.

**Overriding and extension**

If we look again at the `EllipseMorph` class hierarchy in Figure 1.2, we see that the classes `Morph` and `EllipseMorph` both implement `defaultColor`. In fact, if we open a new morph (`Morph new openInWorld`) we see that we get a blue morph, whereas an ellipse will be yellow by default.

We say that `EllipseMorph` overrides the `defaultColor` method that it inherits from `Morph`. The inherited method no longer exists from the point of view of an `Ellipse`.

Sometimes we do not want to override inherited methods, but rather extend them with some new functionality, that is, we would like to be able to invoke the overridden method in addition to the new functionality we are defining in the subclass. In Pharo, as in many object-oriented languages that support single inheritance, this can be done with the help of `super` sends.

A frequent application of this mechanism is in the `initialize` method. Whenever a new instance of a class is initialized, it is critical to also initialize any inherited instance variables. However, the knowledge of how to do this is already captured in the `initialize` methods of each of the superclass in the inheritance chain. The subclass has no business even trying to initialize inherited instance variables!

It is therefore good practice whenever implementing an `initialize` method to send `super initialize` before performing any further initialization:

```
BorderedMorph >> initialize
  "initialize the state of the receiver"
  super initialize.
```
We need super sends to compose inherited behaviour that would otherwise be overridden.

Important It is a good practice that an initialize method start by sending super initialize.

Self sends and super sends

self represents the receiver of the message and the lookup of the method starts in the class of the receiver. Now what is super? super is not the superclass! It is a common and natural mistake to think this. It is also a mistake to think that lookup starts in the superclass of the class of the receiver.

Important self represents the receiver of the message and the method lookup starts in the class of the receiver.

How do self sends differ from super sends?

Like self, super represents the receiver of the message. Yes you read it well! The only thing that changes is the method lookup. Instead of lookup starting in the class of the receiver, it starts in the superclass of the class of the method where the super send occurs.

Important super represents the receiver of the message and the method lookup starts in the superclass of the class of the method where the super send occurs.

We shall see with the following example precisely how this works. Imagine that we define the following three methods:

First we define the method fullPrintOn: on class Morph that just adds to the stream the name of the class followed by the string ’new’ - the idea is that we could execute the resulting string and gets back an instance similar to the receiver.

```plaintext
Morph >> fullPrintOn: aStream
   aStream nextPutAll: self class name, ' new'
```

Second we define the method constructorString that send the message fullPrintOn:.

```plaintext
Morph >> constructorString
   ^ String streamContents: [ :s | self fullPrintOn: s ].
```

Finally, we define the method fullPrintOn: on the class BorderedMorph superclass of EllipseMorph. This new method extends the superclass behavior: it invokes it and adds extra behavior.
Figure 1.3: self and super sends

```
BorderedMorph >> fullPrintOn: aStream
    aStream nextPutAll: '('.
    super fullPrintOn: aStream.
    aStream
        nextPutAll: ')' setBorderWidth: ';
        print: borderWidth;
        nextPutAll: ' borderColor: ', (self colorString: borderColor)
```

Consider the message constructorString sent to an instance of Ellipse-Morph:

```
EllipseMorph new constructorString
>>> '(EllipseMorph new) setBorderWidth: 1 borderColor: Color black'
```

How exactly is this result obtained through a combination of self and super sends? First, anEllipse constructorString will cause the method constructorString to be found in the class Morph, as shown in Figure 1.3.

The method Morph>>constructorString performs a self send of fullPrintOn:. The message fullPrintOn: is looked up starting in the class EllipseMorph, and the method BorderedMorph>>fullPrintOn: is found in BorderedMorph (see Figure 1.3). What is critical to notice is that the self send causes the method lookup to start again in the class of the receiver, namely the class of anEllipse.

At this point, BorderedMorph>>fullPrintOn: does a super send to extend the fullPrintOn: behaviour it inherits from its superclass. Because this is a super send, the lookup now starts in the superclass of the class where the super send occurs, namely in Morph. We then immediately find and evaluate Morph>>fullPrintOn::
Stepping back

A self send is dynamic in the sense that by looking at the method containing it, we cannot predict which method will be executed. Indeed an instance of a subclass may receive the message containing the self expression and redefine the method in that subclass. Here EllipseMorph could redefine the method fullPrintOn: and this method would be executed by method constructorString. Note that by only looking at the method constructorString, we cannot predict which fullPrintOn: method (either the one of EllipseMorph, BorderedMorph, or Morph) will be executed when executing the method constructorString, since it depends on the receiver the constructorString message.

Important A self send triggers a method lookup starting in the class of the receiver. A self send is dynamic in the sense that by looking at the method containing it, we cannot predict which method will be executed.

Note that the super lookup did not start in the superclass of the receiver. This would have caused lookup to start from BorderedMorph, resulting in an infinite loop!

If you think carefully about super send and Figure 1.3, you will realize that super bindings are static: all that matters is the class in which the text of the super send is found. By contrast, the meaning of self is dynamic: it always represents the receiver of the currently executing message. This means that all messages sent to self are looked up by starting in the receiver’s class.

Important A super send triggers a method lookup starting in the superclass of the class of the method performing the super send. We say that super sends are static because just looking at the method we know the class where the lookup should start (the class above the class containing the method).

Message not understood

What happens if the method we are looking for is not found?

Suppose we send the message foo to our ellipse. First the normal method lookup would go through the inheritance chain all the way up to Object (or rather ProtoObject) looking for this method. When this method is not found, the virtual machine will cause the object to send self doesNotUnderstand: #foo. (See Figure 1.4.)

Now, this is a perfectly ordinary, dynamic message send, so the lookup starts again from the class EllipseMorph, but this time searching for the method doesNotUnderstand:. As it turns out, Object implements doesNotUnderstand:. This method will create a new MessageNotUnderstood object which is capable of starting a Debugger in the current execution context.
1.9 Shared variables

Why do we take this convoluted path to handle such an obvious error? Well, this offers developers an easy way to intercept such errors and take alternative action. One could easily override the method `Object>>doesNotUnderstand:` in any subclass of `Object` and provide a different way of handling the error.

In fact, this can be an easy way to implement automatic delegation of messages from one object to another. A `Delegator` object could simply delegate all messages it does not understand to another object whose responsibility it is to handle them, or raise an error itself!

1.9 Shared variables

Now we will look at an aspect of Pharo that is not so easily covered by our five rules: shared variables.

Pharo provides three kinds of shared variables:

1. **Globally** shared variables.

2. **Class variables**: variables shared between instances and classes. (Not to be confused with class instance variables, discussed earlier).

3. **Pool variables**: variables shared amongst a group of classes.

The names of all of these shared variables start with a capital letter, to warn us that they are indeed shared between multiple objects.

**Global variables**

In Pharo, all global variables are stored in a namespace called `Smalltalk`, which is implemented as an instance of the class `SystemDictionary`. Global variables are accessible everywhere. Every class is named by a global variable. In addition, a few globals are used to name special or commonly useful objects.
The variable Processor names an instance of ProcessScheduler, the main process scheduler of Pharo.

```
Processor class
>>> ProcessorScheduler
```

Other useful global variables

**Smalltalk** is the instance of SmalltalkImage. It contains many functionality to manage the system. In particular it holds a reference to the main namespace Smalltalk globals. This namespace includes Smalltalk itself since it is a global variable. The keys to this namespace are the symbols that name the global objects in Pharo code. So, for example:

```
Smalltalk globals at: #Boolean
>>> Boolean

Since Smalltalk is itself a global variable:

```
Smalltalk globals at: #Smalltalk
>>> Smalltalk

(Smalltalk globals at: #Smalltalk) == Smalltalk
>>> true
```

**World** is an instance of PasteUpMorph that represents the screen. World bounds answers a rectangle that defines the whole screen space; all Morphs on the screen are submorphs of World.

**ActiveHand** is the current instance of HandMorph, the graphical representation of the cursor. ActiveHand’s submorphs hold anything being dragged by the mouse.

**Undeclared** is another dictionary, which contains all the undeclared variables. If you write a method that references an undeclared variable, the browser will normally prompt you to declare it, for example as a global or as an instance variable of the class. However, if you later delete the declaration, the code will then reference an undeclared variable. Inspecting Undeclared can sometimes help explain strange behaviour!

Using globals in your code

The recommended practice is to strictly limit the use of global variables. It is usually better to use class instance variables or class variables, and to provide class methods to access them. Indeed, if Pharo were to be implemented from scratch today, most of the global variables that are not classes would be replaced by singletons.

The usual way to define a global is just to perform Do it on an assignment to a capitalized but undeclared identifier. The parser will then offer to declare the
1.9 Shared variables

Figure 1.5: Instance and class methods accessing different variables

global for you. If you want to define a global programmatically, just execute Smalltalk globals at: #AGlobalName put: nil. To remove it, execute Smalltalk globals removeKey: #AGlobalName.

Class variables

Sometimes we need to share some data amongst all the instances of a class and the class itself. This is possible using class variables. The term class variable indicates that the lifetime of the variable is the same as that of the class. However, what the term does not convey is that these variables are shared amongst all the instances of a class as well as the class itself, as shown in Figure 1.5. Indeed, a better name would have been shared variables since this expresses more clearly their role, and also warns of the danger of using them, particularly if they are modified.

In Figure 1.5 we see that rgb and cachedDepth are instance variables of Color, hence only accessible to instances of Color. We also see that superclass, subclass, methodDict and so on are class instance variables, i.e., instance variables only accessible to the Color class.

But we can also see something new: ColorRegistry and CachedColormaps are class variables defined for Color. The capitalization of these variables gives us a hint that they are shared. In fact, not only may all instances of Color access these shared variables, but also the Color class itself, and any of its subclasses. Both instance methods and class methods can access these shared variables.

A class variable is declared in the class definition template. For example, the class Color defines a large number of class variables to speed up color creation; its definition is shown below.

```smalltalk
Object subclass: #Color
  instanceVariableNames: 'rgb cachedDepth cachedBitPattern alpha'
  classVariableNames: 'BlueShift CachedColormaps ColorRegistry ComponentMask GrayToIndexMap GreenShift HalfComponentMask'
```
The class variable ColorRegistry is an instance of IdentityDictionary containing the frequently-used colors, referenced by name. This dictionary is shared by all the instances of Color, as well as the class itself. It is accessible from all the instance and class methods.

Class initialization

The presence of class variables raises the question: how do we initialize them?

One solution is lazy initialization (discussed earlier in this chapter). This can be done by introducing an accessor method which, when executed, initializes the variable if it has not yet been initialized. This implies that we must use the accessor all the time and never use the class variable directly. This furthermore imposes the cost of the accessor send and the initialization test. It also arguably defeats the point of using a class variable, since in fact it is no longer shared.

Another solution is to override the class method initialize (we’ve seen this before in the Dog example).

```
Color class >> initialize
...
self initializeColorRegistry.
...
```

If you adopt this solution, you will need to remember to invoke the initialize method after you define it (by evaluating Color initialize). Although class side initialize methods are executed automatically when code is loaded into memory (from a Monticello repository, for example), they are not executed automatically when they are first typed into the browser and compiled, or when they are edited and re-compiled.

Pool variables

Pool variables are variables that are shared between several classes that may not be related by inheritance. Pool variables were originally stored in pool dictionaries; now they should be defined as class variables of dedicated classes (subclasses of SharedPool). Our advice is to avoid them; you will need them only in rare and specific circumstances. Our goal here is therefore to explain pool variables just enough so that you can understand them when you are reading code.

A class that accesses a pool variable must mention the pool in its class definition. For example, the class Text indicates that it is using the pool dictionary TextConstants, which contains all the text constants such as CR and LF. This
1.10 Internal object implementation note

dictionary has a key #CR that is bound to the value Character cr, i.e., the carriage return character.

ArrayedCollection subclass: #Text
  instanceVariableNames: 'string runs'
  classVariableNames: '
  poolDictionaries: 'TextConstants'
  package: 'Collections-Text'

This allows methods of the class Text to access the keys of the dictionary in the method body directly, i.e., by using variable syntax rather than an explicit dictionary lookup. For example, we can write the following method.

Text >> testCR
  ^ CR == Character cr

Once again, we recommend that you avoid the use of pool variables and pool dictionaries.

1.10 Internal object implementation note

Here is an implementation note for people that really want to go deep inside the way Pharo represents internally objects. The implementation distinguished between two different kinds of objects: # Objects with zero or more fields that are passed by reference and exist on the Pharo heap. # Immediate objects that are passed by value. Depending on version, these are a range of the integers called SmallInteger, all Character objects and possibly a sub-range of 64-bit floating-point numbers called SmallFloat64. In the implementation, such immediate objects occupy an object pointer, most of whose bits encode the immediate’s value and some of the bits encode the object’s class.

The first kind of object, an ordinary object, comes in a number of varieties:

1. Normal objects that have zero or more named instance variables, such as Point which has an x and a y instance variable. Each instance variable holds an object pointer, which can be a reference to another ordinary object or an immediate.

2. Indexable objects like arrays that have zero or more indexed instance variables numbered from 1 to N. Each indexed instance variable holds an object pointer, which can be a reference to another ordinary object or an immediate. Indexable objects are accessed using the messages at: and at:put:. For example ((Array new: 1) at: 1 put: 2; at: 1) answers 2.

3. Objects like Closure or Context that have both named instance variables and indexed instance variables. In the object, the indexed instance variables follow the named instance variables.
4. Objects like ByteString or Bitmap that have indexed instance variables numbered from 1 to N that contain raw data. Each datum may occupy 8, 16 or 32-bits, depending on its class definition. The data can be accessed as either integers, characters or floating-point numbers, depending on how methods at: and at:put: are implemented. The at: and at:put: methods convert between Pharo objects and raw data, hiding the internal representation, but allowing the system to represent efficiently data such as strings, and bitmaps.

The beauty of Pharo is that you normally don’t need to care about the differences between these three kinds of object.

1.11 Chapter summary

The object model of Pharo is both simple and uniform. Everything is an object, and pretty much everything happens by sending messages.

- Everything is an object. Primitive entities like integers are objects, but also classes are first-class objects.

- Every object is an instance of a class. Classes define the structure of their instances via private instance variables and the behaviour of their instances via public methods. Each class is the unique instance of its metaclass. Class variables are private variables shared by the class and all the instances of the class. Classes cannot directly access instance variables of their instances, and instances cannot access instance variables of their class. Accessors must be defined if this is needed.

- Every class has a superclass. The root of the single inheritance hierarchy is ProtoObject. Classes you define, however, should normally inherit from Object or its subclasses. There is no syntax for defining abstract classes. An abstract class is simply a class with an abstract method (one whose implementation consists of the expression self subclassResponsibility). Although Pharo supports only single inheritance, it is easy to share implementations of methods by packaging them as traits.

- Everything happens by sending messages. We do not call methods, we send messages. The receiver then chooses its own method for responding to the message.

- Method lookup follows the inheritance chain; self sends are dynamic and start the method lookup in the class of the receiver, whereas super sends start the method lookup in the superclass of class in which the super send is written. From this perspective super sends are more static than self sends.

- There are three kinds of shared variables. Global variables are accessible everywhere in the system. Class variables are shared between a class, its subclasses and its instances. Pool variables are shared between a
selected set of classes. You should avoid shared variables as much as possible.