As we saw in preceding chapters, in Pharo, everything is an object, and every object is an instance of a class. Classes are no exception: classes are objects, and class objects are instances of other classes. This object model captures the essence of object-oriented programming, and is lean, simple, elegant and uniform. However, the implications of this uniformity may confuse newcomers.

Note that you do not need to fully understand the implications of this uniformity to program fluently in Pharo. Nevertheless, the goal of this chapter is twofold: (1) go as deep as possible and (2) show that there is nothing complex, magic or special here: just simple rules applied uniformly. By following these rules you can always understand why the situation is the way that it is.

1.1 Rules for classes and metaclasses

The Pharo object model is based on a limited number of concepts applied uniformly. To refresh your memory, here are the rules of the object model that we explored in Chapter 1: The Pharo Object Model.

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 follows the inheritance chain.

As we mentioned in the introduction to this chapter, a consequence of Rule 1 is that classes are objects too, so Rule 2 tells us that classes must also be in-
stances of classes. The class of a class is called a metaclass. A metaclass is created automatically for you whenever you create a class. Most of the time you do not need to care or think about metaclasses. However, every time that you use the browser to browse the class side of a class, it is helpful to recall that you are actually browsing a different class. A class and its metaclass are two separate classes, even though the former is an instance of the latter.

To properly explain classes and metaclasses, we need to extend the rules from Chapter : The Pharo Object Model with the following additional rules.

**Rule 6** Every class is an instance of a metaclass.

**Rule 7** The metaclass hierarchy parallels the class hierarchy.

**Rule 8** Every metaclass inherits from Class and Behavior.

**Rule 9** Every metaclass is an instance of Metaclass.

**Rule 10** The metaclass of Metaclass is an instance of Metaclass.

Together, these 10 rules complete Pharo's object model.

We will first briefly revisit the 5 rules from Chapter : The Pharo Object Model with a small example. Then we will take a closer look at the new rules, using the same example.

### 1.2 Revisiting the Pharo object model

**Rule 1.** Since everything is an object, an ordered collection in Pharo is also an object.
Every class is an instance of a metaclass

As we mentioned earlier in Section 1.1, classes whose instances are themselves classes are called metaclasses.

Metaclasses are implicit

Metaclasses are automatically created when you define a class. We say that they are \textit{implicit} since as a programmer you never have to worry about them.
An implicit metaclass is created for each class you create, so each metaclass has only a single instance.

Whereas ordinary classes are named, metaclasses are anonymous. However, we can always refer to them through the class that is their instance. The class of SortedCollection, for instance, is SortedCollection class, and the class of Object is Object class:

```
SortedCollection class
>>> SortedCollection class

Object class
>>> Object class
```

In fact metaclasses are not truly anonymous, their name is deduced from the one of their single instance.

```
SortedCollection class name
>>> 'SortedCollection class'
```

Figure 1.2 shows how each class is an instance of its metaclass. Note that we only skip SequenceableCollection and Collection from the figures and explanation due to space constraints. Their absence does not change the overall meaning.

Querying Metaclasses

The fact that classes are also objects makes it easy for us to query them by sending messages. Let’s have a look:

```
OrderedCollection subclasses
>>> {SortedCollection . ObjectFinalizerCollection .
    WeakOrderedCollection . OCLiteralList . GLMMultiValue}

SortedCollection subclasses
>>> #()
```
1.4 The metaclass hierarchy parallels the class hierarchy

Figure 1.3: The metaclass hierarchy parallels the class hierarchy (elided).

SortedCollection allSuperclasses
>>> an OrderedCollection(OrderedCollection SequenceableCollection
Collection Object ProtoObject)

SortedCollection instVarNames
>>> #('sortBlock')

SortedCollection allInstVarNames
>>> #('array' 'firstIndex' 'lastIndex' 'sortBlock')

SortedCollection selectors
>>> #(#indexForInserting: #sort:to: #addAll: #reSort #sortBlock:
#copyEmpty #addFirst: #insert:before: #defaultSort:to: #median
#at:put: #add: #= #collect: #flatCollect: #sort: #join: #sortBlock)

1.4 The metaclass hierarchy parallels the class hierarchy

Rule 7 says that the superclass of a metaclass cannot be an arbitrary class: it
is constrained to be the metaclass of the superclass of the metaclass’s unique
instance.

SortedCollection class superclass
>>> OrderedCollection class

SortedCollection superclass class
>>> OrderedCollection class

This is what we mean by the metaclass hierarchy being parallel to the class hi-
erarchy. Figure 1.3 shows how this works in the SortedCollection hierarchy.

SortedCollection class
>>> SortedCollection class

SortedCollection class superclass
>>> OrderedCollection class

SortedCollection class superclass superclass
>>> SequenceableCollection class
Uniformity between Classes and Objects

It is interesting to step back a moment and realize that there is no difference between sending a message to an object and to a class. In both cases the search for the corresponding method starts in the class of the receiver, and proceeds up the inheritance chain.

Thus, messages sent to classes must follow the metaclass inheritance chain. Consider, for example, the method `withAll:`, which is implemented on the class side of `Collection`. When we send the message `withAll:` to the class `OrderedCollection`, then it is looked up the same way as any other message. The lookup starts in `OrderedCollection` class (since it starts in the class of the receiver and the receiver is `OrderedCollection`), and proceeds up the metaclass hierarchy until it is found in `Collection` class (see Figure 1.4). It returns a new instance of `OrderedCollection`.

Only one method lookup

Thus we see that there is one uniform kind of method lookup in Pharo. Classes are just objects, and behave like any other objects. Classes have the power to create new instances only because classes happen to respond to the message `new`, and because the method for `new` knows how to create new instances. Normally, non-class objects do not understand this message, but if you have a good reason to do so, there is nothing stopping you from adding a `new` method to a non-metaclass.
1.5 Every metaclass inherits from Class and Behavior

Every metaclass is a kind of a class (a class with a single instance), hence inherits from Class. Class in turn inherits from its superclasses, ClassDescription and Behavior. Since everything in Pharo is an object, these classes all inherit eventually from Object. We can see the complete picture in Figure 1.6.

Where is new defined?

To understand the importance of the fact that metaclasses inherit from Class and Behavior, it helps to ask where new is defined and how it is found. When the message new is sent to a class, it is looked up in its metaclass chain and ultimately in its superclasses Class, ClassDescription and Behavior as shown in Figure 1.7.

Figure 1.5: Classes are objects too.
The question *Where is new defined?* is crucial. `new` is first defined in the class `Behavior`, and it can be redefined in its subclasses, including any of the metaclass of the classes we define, when this is necessary. Now when a message `new` is sent to a class it is looked up, as usual, in the metaclass of this class, continuing up the superclass chain right up to the class `Behavior`, if it has not been redefined along the way.

Note that the result of sending `SortedCollection new` is an instance of `SortedCollection` and *not* of `Behavior`, even though the method is looked up in the class `Behavior`! `new` always returns an instance of `self`, the class that receives the message, even if it is implemented in another class.

```ruby
SortedCollection new class
>>> SortedCollection "not Behavior!"
```

**Common mistake.** A common mistake is to look for `new` in the superclass of the receiving class. The same holds for `new:`, the standard message to create an object of a given size. For example, `Array new: 4` creates an array of 4 elements. You will not find this method defined in `Array` or any of its superclasses. Instead you should look in `Array class` and its superclasses, since that is where the lookup will start (See Figure 1.7).
1.5 Every metaclass inherits from Class and Behavior

Responsibilities of Behavior, ClassDescription, and Class

Behavior provides the minimum state necessary for objects that have instances, which includes a superclass link, a method dictionary and the class format. The class format is an integer that encodes the pointer/non-pointer distinction, compact/non-compact class distinction, and basic size of instances. Behavior inherits from Object, so it, and all of its subclasses, can behave like objects.

Behavior is also the basic interface to the compiler. It provides methods for creating a method dictionary, compiling methods, creating instances (i.e., new, basicNew, new:, and basicNew:), manipulating the class hierarchy (i.e., superclass:, addSubclass:), accessing methods (i.e., selectors, allSelectors, compiledMethodAt:), accessing instances and variables (i.e., allInstances, instVarNames...), accessing the class hierarchy (i.e., superclass, subclasses) and querying (i.e., hasMethods, includesSelector, canUnderstand:, inheritsFrom:, isVariable).

ClassDescription is an abstract class that provides facilities needed by its two direct subclasses, Class and Metaclass. ClassDescription adds a number of facilities to the base provided by Behavior: named instance variables, the categorization of methods into protocols, the maintenance of change sets and the logging of changes, and most of the mechanisms needed for filing out changes.

Class represents the common behaviour of all classes. It provides a class name, compilation methods, method storage, and instance variables. It pro-
Every metaclass is a Metaclass.

1.6 Every metaclass is an instance of Metaclass

One question left is since metaclasses are objects too, they should be instances of another class, but which one? Metaclasses are objects too; they are instances of the class Metaclass as shown in Figure 1.8. The instances of class Metaclass are the anonymous metaclasses, each of which has exactly one instance, which is a class.

Metaclass represents common metaclass behaviour. It provides methods for instance creation (subclassOf:), creating initialized instances of the metaclass's sole instance, initialization of class variables, metaclass instance, method compilation, and class information (inheritance links, instance variables, etc.).

1.7 The metaclass of Metaclass is an instance of Metaclass

The final question to be answered is: what is the class of Metaclass class? The answer is simple: it is a metaclass, so it must be an instance of Metaclass, just like all the other metaclasses in the system (see Figure 1.9).
1.7 The metaclass of `Metaclass` is an instance of `Metaclass`

![Diagram showing the class hierarchy in Pharo]

**Figure 1.9:** All metaclasses are instances of the class `Metaclass`, even the metaclass of `Metaclass`.

Figure 1.9 shows how all metaclasses are instances of `Metaclass`, including the metaclass of `Metaclass` itself. If you compare Figures 1.8 and 1.9 you will see how the metaclass hierarchy perfectly mirrors the class hierarchy, all the way up to `Object` class.

The following examples show us how we can query the class hierarchy to demonstrate that Figure 1.9 is correct. (Actually, you will see that we told a white lie — `Object` class superclass --> `ProtoObject` class, not `Class`. In Pharo, we must go one superclass higher to reach `Class`.)

```
Collection superclass
>>> Object

Collection class superclass
>>> Object class
Object class superclass superclass
>>> Class "NB: skip ProtoObject class"
Class superclass
>>> ClassDescription
ClassDescription superclass
>>> Behavior
Behavior superclass
>>> Object

Collection class class
>>> Metaclass
```
1.8 Chapter summary

This chapter gave an in-depth look into the uniform object model, and a more thorough explanation of how classes are organized. If you get lost or confused, you should always remember that message passing is the key: you look for the method in the class of the receiver. This works on any receiver. If the method is not found in the class of the receiver, it is looked up in its superclasses.

- Every class is an instance of a metaclass. Metaclasses are implicit. A metaclass is created automatically when you create the class that is its sole instance. A metaclass is simply a class whose unique instance is a class.

- The metaclass hierarchy parallels the class hierarchy. Method lookup for classes parallels method lookup for ordinary objects, and follows the metaclass’s superclass chain.

- Every metaclass inherits from Class and Behavior. Every class is a Class. Since metaclasses are classes too, they must also inherit from Class. Behavior provides behaviour common to all entities that have instances.

- Every metaclass is an instance of Metaclass. ClassDescription provides everything that is common to Class and Metaclass.

- The metaclass of Metaclass is an instance of Metaclass. The instance-of relation forms a closed loop, so Metaclass class class is Metaclass.