Chapter 1

Classes and metaclasses

As we saw in Chapter **??**, 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: it is lean, simple, elegant and uniform. However, the implications of this uniformity may confuse newcomers. The goal of this chapter is to 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 **??**.

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 instances 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 **??** 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 Smalltalk's object model.

We will first briefly revisit the 5 rules from Chapter **??** with a small example. Then we will take a closer look at the new rules, using the same example.

1.2 Revisiting the Smalltalk object model

Since everything is an object, an ordered collection in Pharo is also an object.

```
OrderedCollection withAll: #(4 5 6 1 2 3)
    ⟶  an OrderedCollection(4 5 6 1 2 3)
```

Every object is an instance of a class. The class of an ordered collection is the class OrderedCollection:

```
(OrderedCollection withAll: #(4 5 6 1 2 3)) class
    ⟶  OrderedCollection
```

Interestingly, if we send the message asSortedCollection, we get an instance of a different class, namely SortedCollection:

```
((OrderedCollection withAll: #(4 5 6 1 2 3)) asSortedCollection class
    ⟶  SortedCollection
```

By Rule 3, every class has a superclass. The superclass of SortedCollection is OrderedCollection, and the superclass of OrderedCollection is SequenceableCollection:

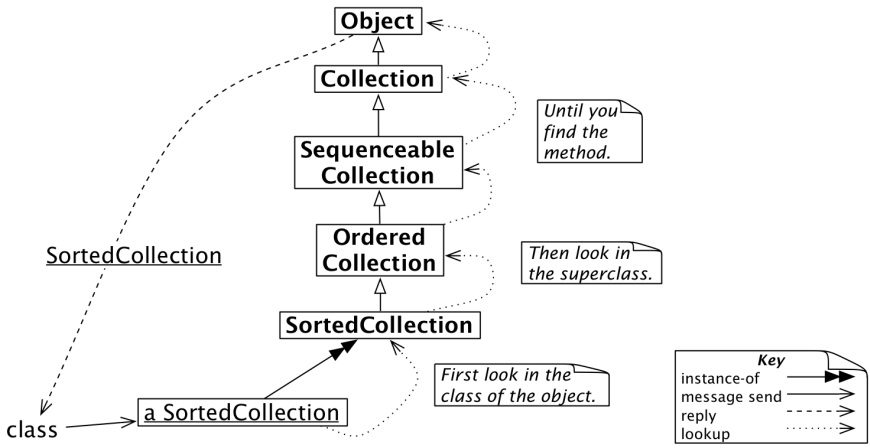

Figure 1.1: Sending a message to a sorted collection.

```
SortedCollection superclass ⟶ OrderedCollection
OrderedCollection superclass ⟶ SequenceableCollection
SequenceableCollection ⟶ Collection
Collection ⟶ Object
```

```
(OrderedCollection withAll: #(4 5 6 1 2 3)) at: 1
    ⟶ 4
```

Everything happens by sending messagess (Rule 4), so we can deduce that withAll: is a message to OrderedCollection, class and asSortedCollection are messages sent to the ordered collection instance, and superclass is a message to OrderedCollection and SortedCollection. The receiver in each case is an object, since everything is an object, but some of these objects are also classes.

Method lookup follows the inheritance chain (Rule 5), so when we send the message class to the result of (OrderedCollection withAll: #(4 5 6 1 2 3)) asSortedCollection, the message is handled when the corresponding method is found in the class Object, as shown in Figure 1.1.

The figure captures the essence of the *is-a* relationship. Our sortedCollection object *is a* SortedCollection instance, but we can also say that it *is a* OrderedCollection and that it *is an* Object, since it responds to the messages defined in all of these classes. In fact, there is a message, isKindOf:, that you can send to any object to find out if it is in an *is a* relationship with a given class:

```
sortedCollection := ((OrderedCollection withAll: #(4 5 6 1 2 3)) asSortedCollection.
sortedCollection isKindOf: SortedCollection ⟶ true
```

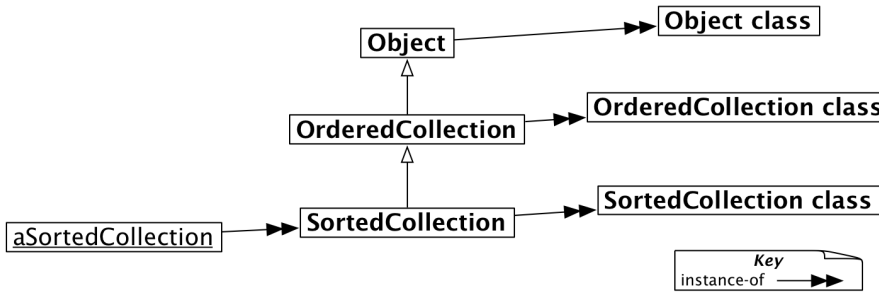

Figure 1.2: The metaclasses of SortedCollection and its superclasses.

```
sortedCollection isKindOf: OrderedCollection    ⟶    true
sortedCollection isKindOf: Object               ⟶    true
```

1.3 Every class is an instance of a metaclass

As we mentioned 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 *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 by global variables, 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    ⟶    Color class
Object class    ⟶    Object class
```

Figure 1.2 shows how each class is an instance of its (anonymous) metaclass. Note that space reason we skip SequenceableCollection and Collection from the figures and explanation for now. It does not change anything to contents.

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}
```

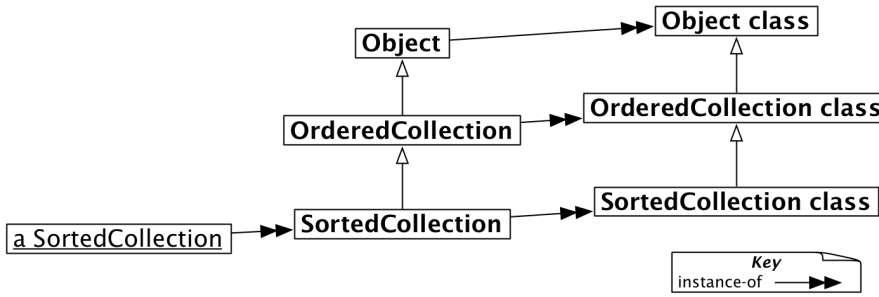

Figure 1.3: The metaclass hierarchy parallels the class hierarchy.

```
SortedCollection subclasses ⟶ #()
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 hierarchy; 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
SortedCollection class superclass superclass superclass superclass ⟶ Object
    class
```

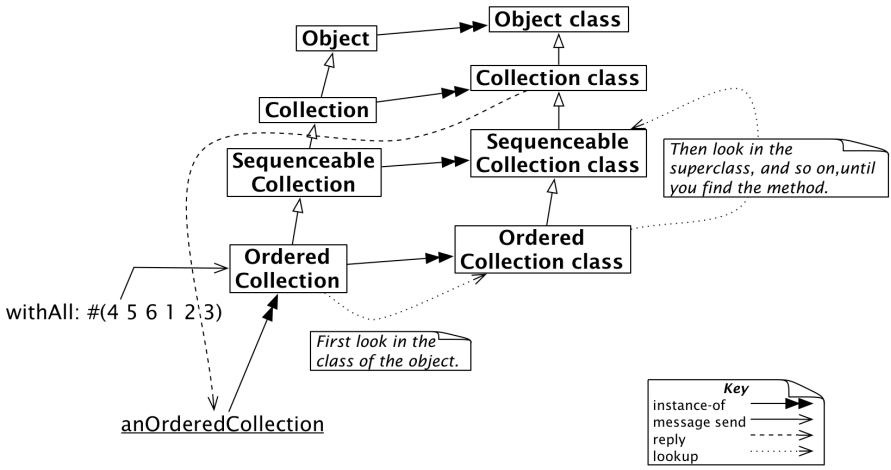

Figure 1.4: Message lookup for classes is the same as for ordinary objects.

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. If we send the message withAll: to OrderedCollection, then it will be looked-up the same way as any other message. The lookup starts in OrderedCollection class, and proceeds up the metaclass hierarchy until it is found in Collection class (see Figure 1.4).

```
(OrderedCollection withAll: #(4 5 6 1 2 3)    ⟶    an OrderedCollection (4 5 6 1 2
    3)
```

Note that we get as a result an ordinary Color blue, and not a translucent one — there is no magic!

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.

Since classes are objects, we can also inspect them.

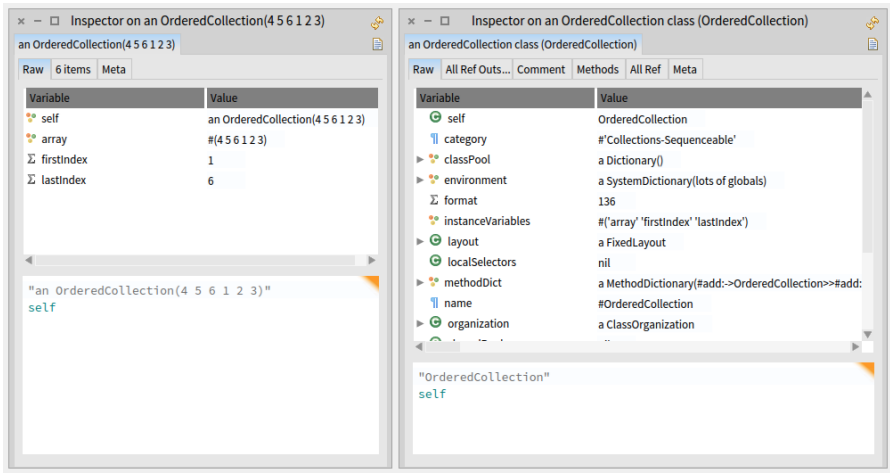

Figure 1.5: Classes are objects too.

Inspect OrderedCollection withAll: #(4 5 6 1 2 3) and OrderedCollection.

Notice that in one case you are inspecting an instance of OrderedCollection and in the other case the OrderedCollection class itself. This can be a bit confusing, because the title bar of the inspector names the *class* of the object being inspected.

The inspector on OrderedCollection allows you to see the superclass, instance variables, method dictionary, and so on, of the OrderedCollection class, as shown in Figure 1.5.

1.5 Every metaclass Inherits from Class and Behavior

Every metaclass *is-a* class, 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.

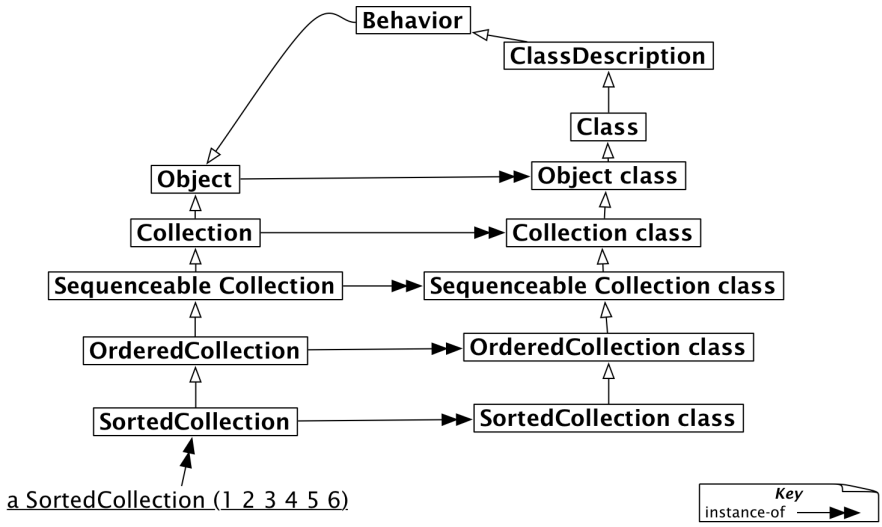

Figure 1.6: Metaclasses inherit from Class and Behavior.

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.

```
SortedCollection new class    ⟶    SortedCollection    "not Behavior!"
```

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.

Responsibilities of Behavior, ClassDescription and Class. Behavior provides the minimum state necessary for objects that have instances: this includes a superclass link, a method dictionary and the class format. The class format is an integer that encodes the pointer/non-pointer distinction,

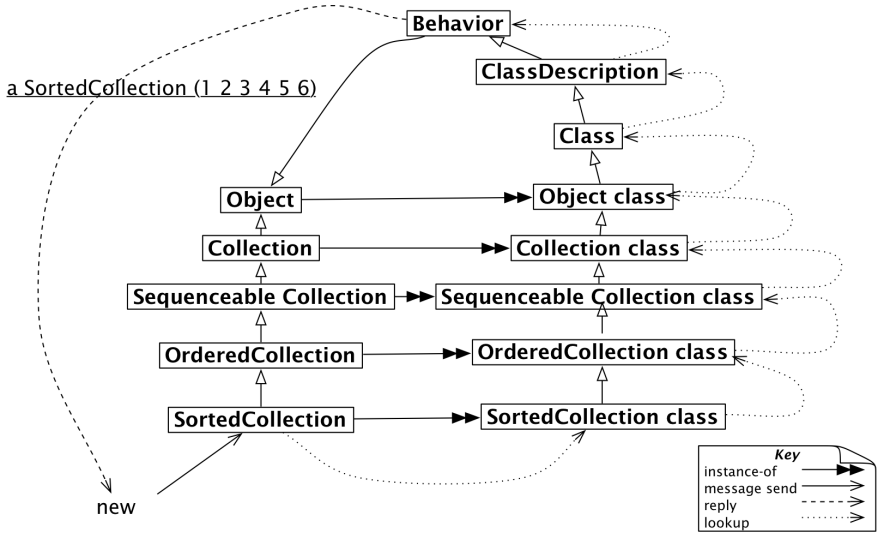

Figure 1.7: new is an ordinary message looked up in the metaclass chain.

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 basis 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 provides a concrete representation for class variable names and shared pool variables (addClassVarName:, addSharedPool:, initialize). Since a metaclass is a class for its sole instance (*i.e.*,, the non-meta class) and provides the service for the instance as a class, all metaclasses inherit ultimately from Class.

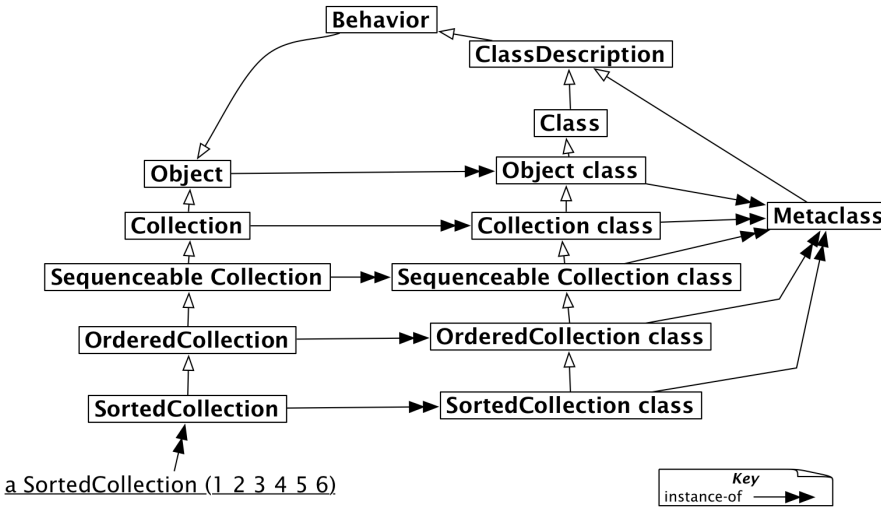

Figure 1.8: Every metaclass is a Metaclass.

1.6 Every metaclass is an instance of Metaclass

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, % (different semantics can be introduced) 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).

The figure 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.

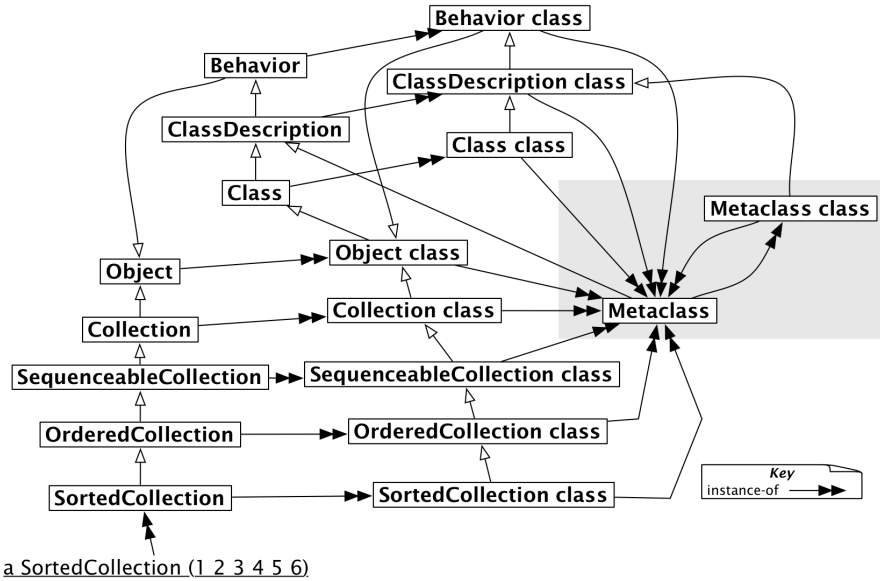

Figure 1.9: All metaclasses are instances of the class Metaclass, even the metaclass of Metaclass.

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.)

Script 1.1: *The class hierarchy*

```
SortedCollection superclass ⟶ OrderedCollection
OrderedCollection superclass ⟶ SequenceableCollection
SequenceableCollection superclass ⟶ Collection
Collection superclass ⟶ Object
```

Script 1.2: *The parallel metaclass hierarchy*

```
SortedCollection class superclass ⟶ OrderedCollection class
OrderedCollection class superclass ⟶ SequenceableCollection
    class
SequenceableCollection class superclass ⟶ Collection class
Collection class superclass ⟶ Object class
Object class superclass superclass ⟶ Class "NB: skip ProtoObject class"
Class superclass ⟶ ClassDescription
ClassDescription superclass ⟶ Behavior
Behavior superclass ⟶ Object
```

Script 1.3: *Instances of Metaclass*

```
SortedCollection class class    ⟶    Metaclass
OrderedCollection class class          ⟶    Metaclass
SequenceableCollection class class          ⟶    Metaclass
Collection class class          ⟶    Metaclass
Object class class              ⟶    Metaclass
Behavior class class            ⟶    Metaclass
```

```
Metaclass class class    ⟶    Metaclass
Metaclass superclass     ⟶    ClassDescription
```

1.8 Chapter summary

Now you should understand better how classes are organized and the impact of a uniform object model. 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.
- 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 –> Metaclass.