9 Collections

9.1 Introduction

Recall that we have defined a collection as a type of container that is traversable, that is, a container that allows access to all its elements. The process of accessing all the elements of a collection is also called **iteration**. Iteration over a collection may be supported in several ways depending on the agent that controls the iteration and where the iteration mechanism resides. In this chapter we examine iteration design alternatives and discuss how collections and iteration work in Ruby. Based on this discussion, we will decide how to support collection iteration in our container hierarchy and how to add collections to the hierarchy.

9.2 Iteration Design Alternatives

There are two ways that iteration may be controlled.

> **Internal iteration**—When a collection controls iteration over its elements, then iteration is said to be *internal*. A client wishing to process each element of a collection packages the process in some way (typically in a function or a block), and passes it to the collection, perhaps with instruction about how iteration is to be done. The collection then applies the processing to each of its elements. This mode of control makes it easier for the client to iterate over a collection, but with less flexibility in dealing with issues that may arise during iteration.
>
> **External iteration**—When a client controls iteration over a collection, the iteration is said to be *external*. In this case, a collection must provide operations that allow an iteration to be initialized, to obtain the current element from the collection, to move on to the next element in the collection, and to determine when iteration is complete. This mode of control imposes a burden on the client in return for more flexibility in dealing with the iteration.

In addition to issues of control, there are also alternatives for where the iteration mechanism resides.

> **In the language**—An iteration mechanism may be built into a language. For example, Java, Visual Basic, and Ruby have special looping control structures that provide means for external iteration over collections. Ruby has a special control structure for yielding control to a block passed to an operation that provides support for internal iteration.

In the collection—An iteration mechanism may reside in a collection. In the case of a collection with an external iteration mechanism, the collection must provide operations to initialize iteration, return the current element, advance to the next element, and indicate when iteration is complete. In the case of a collection with an internal iteration mechanism, the collection must provide an operation that accepts a packaged process and applies it to each of its elements.

In an iterator object—An iteration mechanism may reside in a separate entity whose job is to iterate over an associated collection. In this case the operations mentioned above to support internal or external iteration are in the iterator object and the collection usually has an operation to create iterators.

Combining these design alternatives gives six ways that iteration can be done: internal iteration residing in the language, in the collection, or in an iterator object, and external iteration residing in the language, in the collection, or in an iterator object. Each of these alternatives has advantages and disadvantages, and various languages and systems have incorporated one or more of them. For example, most object-oriented languages have external iteration residing in iterators (this is known as the Iterator design pattern). Nowadays many languages provide external iteration in control structures, as mentioned above. Ruby provides five of the six alternatives! We will now consider the Iterator design pattern, and then iteration in Ruby.

9.3 The Iterator Design Pattern

A software design pattern is an accepted solution to a common design problem that is proposed as a model for solving similar problems.

> **Software design pattern**: A model proposed for imitation in solving a software design problem.

Design patterns occur at many levels of abstraction. For examples, a particular algorithm or data structure is a low-level design pattern, and the overall structure of a very large program (such as a client-server structure) is a high-level design pattern. The **Iterator pattern** is a mid-level design pattern that specifies the composition and interactions of a few classes and interfaces.

The Iterator pattern consists of an `Iterator` class whose instances are created by an associated collection and provided to clients. The `Iterator` instances house an external iteration mechanism. Although `Iterator` class functionality can be packaged in various ways, `Iterator` classes must provide the following functionality.

> *Initialization*—Prepare the `Iterator` object to traverse its associated collection. This operation will set the current element (if there is one).
>
> *Completion Test*—Indicate whether traversal by this `Iterator` is complete.
>
> *Current Element Access*—Provide the current collection element to the client. The precondition for this operation is that iteration is not complete.
>
> *Current Element Advance*—Make the next element in the collection the current element. This operation has no effect once iteration is complete. However, iteration may become complete when it is invoked—in other words, if the current item is the last, executing this operation completes the iteration, and calling it again does nothing.

The class diagram in Figure 1 below presents the static structure of the Iterator pattern. The four operations in the Iterator interface correspond to the four functions listed above. The `iterator()` operation in the `Collection` interface creates and returns a new concrete iterator for the particular collection in which it occurs; this is called a **factory method** because it manufactures a class instance.

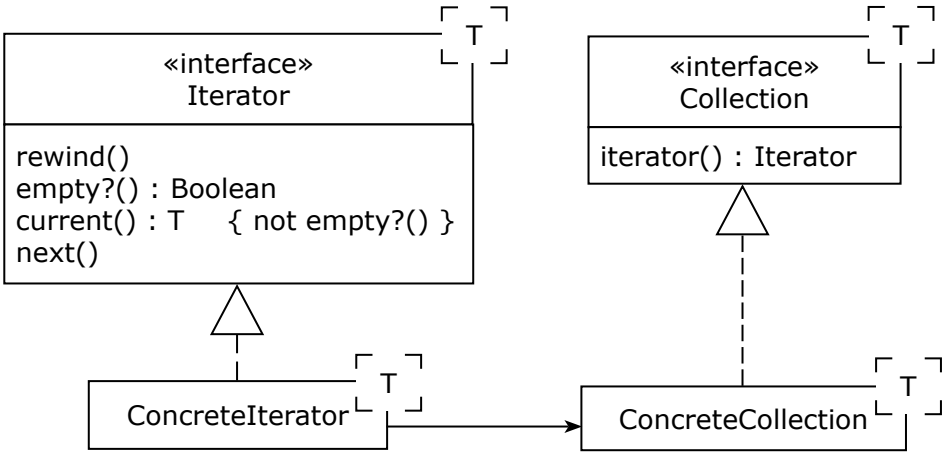

Figure 1: The Iterator Pattern

The interfaces and classes in this pattern are templated with the type of the elements held in the collection. The arrow from the ConcreteIterator to the ConcreteCollection indicates that the ConcreteIterator must have some sort of reference to the collection with which it is associated so that it can access its elements.

A client uses an iterator by asking a ConcreteCollection for one by calling its iterator() operation. The client can then rewind the iterator and use a while loop to access its elements. The pseudocode below in Figure 2 illustrates how this is done.

```
c = ConcreteCollection.new
…
i = c.iterator
i.rewind
while !i.empty?
  element = i.current
  // process element
  i.next
end
```

Figure 2: Using an Iterator

Note that if the programmer decided to switch from one ConcreteCollection to another, only one line of this code would have to be changed: the first. Because of the use of interfaces, the code would still work even though a different ConcreteIterator would be used to access the elements of the collection.

9.4 Iteration in Ruby

As mentioned, Ruby supports five of the six alternatives for doing iteration: there is no support in Ruby for external iteration residing in collections. This is probably because there are so many other ways to iterate over collections that there is no need for this alternative. Lets now consider the five different ways of doing iteration in Ruby.

Internal iteration supported by the language. A block of code may be passed as a special parameter to any Ruby operation, and this operation may then yield control (along with data arguments) to the block. This mechanism, which is not tied to collections but can be used anywhere in the language, can be considered a special form of internal iteration with control residing in the language itself. This is also the way that internal iterators are implemented for collections defined by programmers.

External iteration provided by the language. Ruby has a `for/in` loop that can be used with any collection. Each element of the collection is assigned in turn to a loop variable, and the body of the loop can then access and process the elements of the collection through this loop variable.

Internal iteration in a collection. This is the preferred collection traversal mechanism in Ruby. `Enumerable` is a special mixin module that provides over twenty internal iteration operations. All built-in collections mix in this module (and user-defined collections should as well). The internal iteration operations all accept parameterized blocks as the packaged process and apply the block to each element of the collection.

Internal and external iterator objects. In Ruby, an *enumerator* is an entity whose job is to iterate over a collection; in other words, an enumerator is an iterator object. Enumerators are created from any object that implements the `each()` operation by calling `to_enum()` or `enum_for()`. All enumerators mix in the `Enumerable` module, so they include all the internal collection iterator operations. Hence they are internal iteration objects. Furthermore, they also include operations for external iteration: `rewind()` initializes an enumerator in preparation for iteration and `next()` fetches the next object in the iteration (thus combining the actions of fetching the current element and advancing to the next element). If `next()` is called after iteration is complete, it raises a `StopIteration` exception, which is how iteration is terminated. The `loop` control structure catches this exception and terminates automatically, so explicitly handling exceptions is often not necessary when using an enumerator as an external iterator.

Although Ruby offers so many alternative forms of iteration, it is clearly designed to favor internal iteration. Usually, this is not a problem, but occasionally external iterators are needed to increase control over iteration. Ironically, the external iteration mechanisms in Ruby do not provide much additional flexibility. The `for/in` construct is quite rigid, and enumerators lack a non-exception based means of determining iteration termination. We will enrich our container hierarchy by adding better external iteration facilities.

9.5 Collections, Iterators, and Containers

There are many sorts of collections, including simple linear sequences (lists), unordered aggregates (sets), and aggregates with keyed access (maps). There are not many operations common to this wide variety of collections that should be included in the Collection interface. For example, although one must be able to add elements to every collection, how elements are added varies. Adding an element to an unordered collection simply involves the element added. Adding to a list requires specifying where the element is to be added, and adding to a map requires that the access key be specified.

Two operations do come to mind, however: we may ask of a collection whether it contains some element, and we may compare collections to see whether they are equal. Although the Ruby `Object` class includes equality operations, adding one to the Collection interface will force concrete collections to implement it. We also add a collection containment query operation to the Collection interface.

In the spirit of Ruby, it seems wise to mix the Ruby `Enumerable` module into the Collection interface. This provides a rich variety of internal iterators. Given the drawbacks of internal iteration, it might be advisable to include external iterators based on the Iterator design pattern as well. Hence we include an `iterator()` factory method in the Collection interface.

Figure 3 shows the final Collection interface and its place in the Container hierarchy, along with the Ruby `Enumerable` mixin, which is indicated in UML using a dependency arrow.

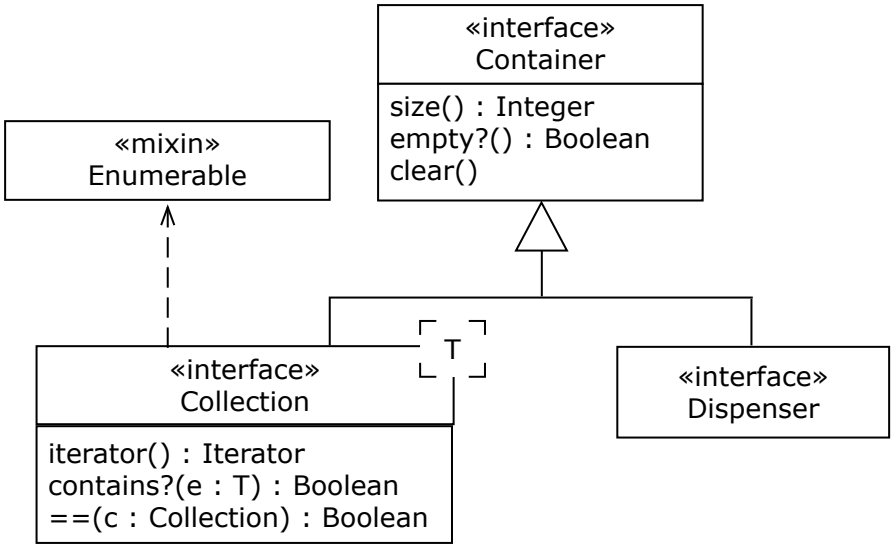

Figure 3: The Collection Interface in the Container Hierarchy

9.6 Summary and Conclusion

Collections are traversable containers and hence require some sort of iteration facility. There are many alternative designs for such a facility. The Iterator design pattern, a way to use external iterator objects, is a powerful possibility, but Ruby favors and supports internal iteration. We will incorporate both Ruby internal iterators and external iterator objects from the Iterator design pattern in our Collection interface.

Collections should also include an equality operation that indicates whether two collections are the same, along with a collection containment operation. Both of these operations appear in our Collection interface.

9.7 Review Questions

1. What are the alternatives for controlling iteration?
2. Where might iteration mechanisms reside?
3. What are the six alternatives for designing collection iteration facilities?
4. What is a software design pattern?
5. What functions must an external Iterator object provide in the Iterator pattern?
6. Which of the six iteration design alternatives does Ruby not support?
7. What is the `Enumerable` module in Ruby?
8. What does the contains?() operation return when a Collection is empty?

9.8 Exercises

1. What is the point of an iterator when each element of a list can already be accessed one by one using indices?
2. Explain why a Dispenser does not have an associated iterator.
3. Java has an iterators, but the Java Iterator interface does not have a rewind() operation. Why not?
4. Would it be possible to have an Iterator interface with only a single operation? If so, how could the four Iterator functions be realized?
5. How might external iterators residing in collections be added to Ruby?
6. Write a generic Ruby implementation of the Collection contains?() operation using
   a) an internal collection iterator
   b) an external iterator object
   c) an internal iterator object
   d) the `for/in` loop construct
7. What happens to an iterator (any iterator) when its associated collection changes during iteration?
8. Consider the problem of checking whether two collections contain the same values. Can this problem be solved using collection internal iterators? Can it be solved using iterator objects?

9.9 Review Question Answers

1. There are two alternatives for controlling iteration: the collection may control it (internal iteration) or the client may control it (external iteration).
2. Iteration mechanisms can reside in three places: in the language (in the form of control structures), in the collection (as a set of operations), or in a separate iterator object (with certain operations).
3. The six alternatives for designing collection iteration facilities are generated by combining control alternatives with residential alternatives, yielding the following six possibilities: (1) internal control residing in the language, (2) external control residing in a language, (3) internal control residing in the collection, (4) external control residing in the collection, (5) internal control residing in an iterator object, (5) external control residing in an iterator object.
4. A software pattern is model proposed for imitation in solving a software design problem. In other words, a pattern is a way of solving a design or implementation problem that has been found to be successful, and that can serve as a template for solving similar problems.
5. An Iterator must provide four functions: a way to initialize the Iterator object to prepare to traverse its associated Collection, a way to fetch the current element of the Collection, a way to advance to the next element of the Collection, and a way to indicate that all elements have been accessed.
6. Ruby does not support external iteration residing in the collection.
7. The `Enumerable` module is used to mix in operations for internal iteration to other classes. It is used to add internal iteration facilities to collections and to enumerators, which are iterator objects.
8. If a Collection is empty, then it contains nothing so the contains?() operation returns false no matter what its argument.