5 Containers

5.1 Introduction

Simple abstract data types are useful for manipulating simple sets of values, like integers or real numbers, but more complex abstract data types are crucial for most applications. A category of complex ADTs that has proven particularly important is containers.

Container: An entity that holds finitely many other entities.

Just as containers like boxes, baskets, bags, pails, cans, drawers, and so forth are important in everyday life, containers such as lists, stacks, and queues are important in programming.

5.2 Varieties of Containers

Various containers have become standard in programming over the years; these are distinguished by three properties:

Structure—Some containers hold elements in some sort of structure, and some do not. Containers with no structure include sets and bags. Containers with linear structure include stacks, queues, and lists. Containers with more complex structures include multidimensional matrices.

Access Restrictions—Structured containers with access restrictions only allow clients to add, remove, and examine elements at certain locations in their structure. For example, a stack only allows element addition, removal, and examination at one end, while lists allow access at any point. A container that allows client access to all its elements is called **traversable**, **enumerable**, or **iterable**.

Keyed Access—A collection may allow its elements to be accessed by keys. For example, maps are unstructured containers that allows their elements to be accessed using keys.

5.3 A Container Taxonomy

It is useful to place containers in a taxonomy to help understand their relationships to one another and as a basis for implementation using a class hierarchy. The root of the taxonomy is Container. A Container may be structured or not, so it cannot make assumptions about element location (for example, there may not be a first or last element in a container). A Container may or may not be accessible by keys, so it cannot make assumptions about element retrieval methods (for example, it cannot have a key-based search method). Finally, a Container may or may not have access restrictions, so it cannot have addition and removal operations (for example, only stacks have a push() operation), or membership operations.

The only things we can say about Containers is that they have some number of elements. Thus a Container can have a size() operation. We can also ask (somewhat redundantly) whether a Container is empty. And although a Container cannot have specific addition and removal operations, it can have an operation for emptying it completely, which we call clear().

A Container is a broad category whose instances are all more specific things; there is never anything that is just a Container. In object-oriented terms, a Container is an interface, not a class. These considerations lead to the UML class diagram in Figure 1 below.

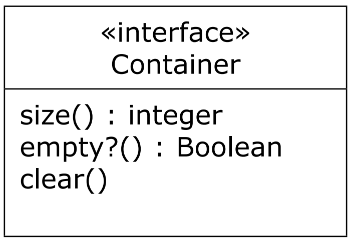

Figure 1: The Container Interface

There are many ways that we could construct our container taxonomy from here; one way that works well is to make a fundamental distinction between traversable and non-traversable containers:

Collection: A traversable container.

Dispenser: A non-traversable container.

Collections include lists, sets, and maps; dispensers include stacks and queues. With this addition, our container hierarchy appears in Figure 2.

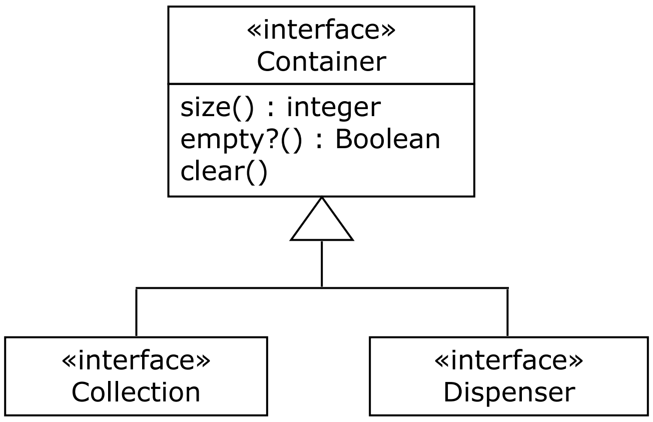

Figure 2: Top of the Container Taxonomy

Dispensers are linearly ordered and have access restrictions. As noted, Dispensers include stacks and queues. We turn in the next chapter to detailed consideration of these non-traversable containers.

5.4 Interfaces in Ruby

Recall that in object-oriented programming, an interface is a collection of abstract operations that cannot be instantiated. Although Ruby is object-oriented, it does not have interfaces. Interface-like classes can be constructed in Ruby by creating classes whose operations have empty bodies. These *pseudo-interface* classes can be used as super-classes of classes that implement the interface. Pseudo-interface classes can still be instantiated, and their operations can still be called without overriding them, so they are only a bit like real interfaces. One way to make them more like interfaces, however, is to implement the operations in the pseudo-interface so that they raise exceptions if they are called. This forces sub-classes to override the operations in these pseudo-interface classes before they can be used.

This is the approach we will take in implementing interfaces in Ruby: we will make super-classes whose operations raise `NotImplementedError` exceptions. Classes implementing a pseudo-interface will then have to inherit from it and override its operations.

5.5 Review Questions

1. Are sets structured? Do sets have access restrictions? Do sets have keyed access?
2. If c is a Container and c.clear() is called, what does c.empty?() return? What does c.size() return?
3. What would happen if you tried to instantiate a pseudo-interface class implemented as suggested above in Ruby?

5.6 Exercises

1. Consider a kind of Container called a Log that is an archive for summaries of transactions. Summaries can be added to the end of a Log, but once appended, they cannot be deleted or changed. When a summary is appended to a Log, it is time-stamped, and summaries can be retrieved from a Log by their time stamps. The summaries in a Log can also be examined in arbitrary order.
   a) Is a Log structured? If so, what kind of structure does a Log have?
   b) Does a Log have access restrictions?
   c) Does a Log provide keyed access? If so, what is the key?
   d) In the container hierarchy, would a Log be a Collection or a Dispenser?

2. Consider a kind of Container called a Shoe used in an automated Baccarat program. When a Shoe instance is created, it contains eight decks of Cards in random order. Cards can be removed one at a time from the front of a Shoe. Cards cannot be placed in a Shoe, modified, or removed from any other spot. No Cards in a Shoe can be examined.
   a) Is a Shoe structured? If so, what kind of structure does a Shoe have?
   b) Does a Shoe have access restrictions?
   c) Does a Shoe provide keyed access? If so, what is the key?
   d) In the container hierarchy, would a Shoe be a Collection or a Dispenser?

3. Consider a kind of Container called a Randomizer used to route packets in an anonymizer. Packets go into the Randomizer at a single input port, and come out randomly at one of *n* output ports, each of which sends packets to different routers. Packets can only go into a Randomizer at the single input port, and can only come out one of the *n* output ports. Packets come out of a single output port in the order they enter a Randomizer. Packets cannot be accessed when they are inside a Randomizer.
   a) Is a Randomizer structured? If so, what kind of structure does a Randomizer have?
   b) Does a Randomizer have access restrictions?
   c) Does a Randomizer provide keyed access? If so, what is the key?
   d) In the container hierarchy, would a Randomizer be a Collection or a Dispenser?

5.7 Review Question Answers

1. Sets are not structured—elements appear in sets or not, they do not have a position or location in the set. Sets do not have access restrictions: elements can be added or removed arbitrarily. Elements in sets do not have keys (they are simply values), so there is no keyed access to elements of a set.
2. When a Container c is cleared, it contains no values, so `c.empty?()` returns true, and `c.size()` returns 0.
3. A pseudo-interface class implemented in Ruby with operations that raise `NotImplementedError` exceptions when they are called could be instantiated without any problem, but such objects could not be used for anything useful. The only thing that such a class would be good for would be as a super-class of classes that override its operations, which is pretty much what an interface is good for as well.