Iteration Abstraction

Learning Goals

You must be able to:

- ✓ **describe how interfaces, classes and objects can be used to provide iteration abstraction (also known as the Iterator design pattern)**
- ✓ **use an iterator to repeat a set of instructions on a collection of objects**
- ✓ **use a for-each loop to iterate over objects in a collection**

Iteration Abstraction

Iteration is about repetition. Often, in object-oriented software, we need to repeat the same set of instructions on a number of different objects. Most interesting software systems involve manipulating collections of objects. Software that helps you organize your digital music library would likely represent each song as a separate object. Software that supports registration in courses at UBC would likely represent each student as a separate object. Software that monitors a chemical plant would likely have separate objects representing each sensor that provides data about the current state of the plant.

Often, a software system will need to perform a set of operations on each object in a collection. Repeating the operations on each object requires iteration across the collection.

You may recall from the Data Abstraction reading that there are three different kinds of abstraction used in the Java programming language: procedural abstraction, data abstraction and iteration abstraction. We covered procedural abstraction and data abstraction in detail in the Data Abstraction and Developing Robust Classes readings. In this reading, we will look at how Java supports iteration abstraction, which supports "iterat[ing] over arbitrary types of data in a convenient and efficient way" [1, p. 125].

Iteration abstraction as it is supported in Java is an example of the *Iterator* design pattern. A design pattern is a reusable solution to a commonly occurring problem. Christopher Alexander introduced the concept of a pattern in (building) architecture. The Gang of Four[1] (Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides) popularized the idea of applying design patterns to software design in the book, Design Patterns: Elements of Reusable Object-oriented Software.

A design pattern (at least in the GoF book) is described, in part, by providing a description of the classes (sometimes called participants) involved in the design and how the classes collaborate. Each design pattern addresses a particular design intent. For example, the Iterator design pattern addresses the need to iterate over collections without knowing details about what objects are maintained by the collection. Over the course of the term, we will look at a few design patterns. In addition to improving design, a benefit of design patterns is that they provide a vocabulary that allows software developers to talk about code at a higher level of discourse. For example, saying Iterator to a software developer versed in design patterns immediately conveys particular ways in which the classes may collaborate without investigating the details of the code. Design patterns have been highly influential since they were introduced in the mid-1990s.

Iteration Abstraction in Java

Let us start our investigation of iteration abstraction in Java by considering a concrete example. We have seen in earlier readings that the Java Collections Framework is a standard part of Java that supports managing a collection of objects. Using the Java Collections Framework, collections of objects can be represented in multiple ways depending upon properties of the collection. For example, we might want to maintain a collection of objects (e.g., emails) in the order in which they were added to a collection (e.g., an email inbox). At other times, we might want to maintain a collection of objects (e.g., contacts in an address box) with the property that no object appears twice (e.g., a set of contacts).

The Java Collections Framework supports the data abstraction of a `Collection` with multiple implementations. The `Collection` interface supports operations common to all collections, such as determining if an element is in the collection or asking if the collection is empty. Here is a partial view of the Collection interface:

[1] Sometimes referred to as GoF.

```
public interface Collection<E> … {
   …
   boolean contains(Object element);
   boolean isEmpty();
}
```

Specific kinds of collections extend this interface[2] to add additional collection-type specific operations. For example, here is a partial view of the `List` interface, which adds such operations as accessing an element at a specific location in the list:

```
public interface List<E> extends Collection<E> {
   …
   E get(int index);
}
```

As you have seen in the course thus far, an interface such as `List` can be implemented in several ways. One implementation of the `List` interface provided in the Java Collections Framework is `ArrayList`, which uses an array to store the list of objects.[3] We will use `ArrayList` as an example in this reading.

Iterator

Conceptually, iteration is something that happens to a collection, not something that is part of a collection. When we iterate across a collection of objects, we want to be able to access each object in the collection in turn, but we need not alter the collection. The `Collection` interface makes this possible by providing an object that knows how to iterate across an instance of a collection. More specifically, the `Collection` interface includes the following operation:

```
public interface Collection<E> … {
   Iterator<E> iterator(); …
}
```

[2] Just as a `class` can extend another `class`, an `interface` in Java can extend another `interface`. The sub-`interface` inherits all operation definitions from its super-`interface`. A `class` implementing the sub-`interface` must provide definitions for all operations defined in the sub-`interface` and the super-`interface`(s). In contrast to a `class` which can extend only one other `class`, an `interface` can extend more than one other `interfaces`.

[3] Conceptually, an array is a set of memory locations in order that can be indexed. At each memory location, an object can be stored.

An `Iterator` is an object that knows how to visit all elements of a collection. Here is a partial specification of the `Iterator` interface:

```
public interface Iterator<E> {
   boolean hasNext();
   E next();
   …
}
```

With `Iterator`, it becomes possible to iterate across a collection without knowing the details of how the collection is represented. For instance, the following code can access each integer stored in an `ArrayList` object in turn to determine the sum of all integers stored in the list.

```
List<Integer> aListOfIntegers = new ArrayList<Integer>();
aListOfIntegers.add(3);
aListOfIntegers.add(5);

int sum= 0;
Iterator<Integer> anIterator = aListOfIntegers.iterator();
while (anIterator.hasNext()) {
   Integer i = anIterator.next();
   sum = sum + i;
}

assert (sum == 8);
```

After the list object is created and populated with two values (3 and 5), we initialize a variable to compute the sum and we retrieve an iterator object from the list. We then loop over the list by asking the iterator (object referred to by the `anIterator` variable) whether any more values are available (`hasNext()`). If there are, we proceed into the body of the `while` loop, and use the `next()` operation on the iterator to get the next Integer from the list. The call to `next()` also advances the iterator to the next element of the list so that when we next call `hasNext()` at the top of the loop, we are seeing whether or not there is again another `Integer` to process. We stop when we have visited every element in the list.

Implementing Iterator

`Iterator` is a data abstraction just as `Collection` is a data abstraction. Just as we must provide implementations of `Collection`, such as `ArrayList`, we must also provide implementations of `Iterator`. In order to traverse a collection of a particular type, an `Iterator` must know details about how the collection is implemented. As a result, there is typically a separate `Iterator` implementation for each kind of `Collection`. For example, an `ArrayListIterator` class might be created to support iterating over an `ArrayList` object. Since `iterator()` is a method on `Collection`, subtypes of

`Collection` must implement this method. Thus, `ArrayList` must provide an implementation of `iterator()`. Assuming that `ArrayListIterator` exists, the implementation of `iterator()` on `ArrayList` might look something like this:

```
… class ArrayList<E> implements List<E> {

   public Iterator<E> iterator() {
       return new ArrayListIterator<E>();
   }
}
```

where `ArrayListIterator` is declared something like:

```
… class ArrayListIterator<E> implements Iterator<E> …
```

Now the code above leaves out a lot of details. For example, `ArrayListIterator` has to have special access to the representation of data on `ArrayList`. These details do not matter for now, but ask if you want to know more.

Iterable

The ability to use an `Iterator` to visit each item in a collection provides iteration abstraction in Java. The current version of Java takes iteration abstraction one step further.

If you look back at the `while` loop above that we wrote to sum up the integers in a collection, you might note that the structure of the `while` loop is generic. The main iteration over the collection is as follows, where `<iterator>` is some `Iterator` object:

```
while (<iterator>.hasNext() ) {
   <element> = <iterator>.next();  // Get current element and
                                   // advance iterator to
                                   // next in list.
   // do some stuff involving element
}
```

The for-each construct you have seen in Java abstracts this structure and looks as follows:

```
for (<type> <variable>: <collection>) { … }
```

When this statement is encountered during execution, each element in `<collection>` is, in turn, assigned to the `<variable>` and the body of the `for` loop executes once per element in `<collection>`. In other words, an iterator is accessed for `<collection>` and the iterator is used to go through each element of the collection in turn. This `for` loop is the same as the `while` loop above. The syntax is easier to type and reduces errors.

One choice the makers of the Java programming language could have made was to limit <collection> to subtypes of the `Collection` interface. Making `Collection` special in this way is not ideal. Instead, a new interface was introduced that just captures the ability to provide an `Iterator`, sometimes known as a generator, for a collection.

```
public interface Iterable<E> {
   Iterator<E> iterator();
}
```

This interface returns an iterator over a set of elements of type `E`. If a class implements this interface, an object of that class can be the target (`<collection>` above) in a for-each construct.

`Collection` is one interface that implements `Iterable`:

```
public interface Collection<E> implements Iterable<E> …
```

You can define your own data abstractions that implement `Iterable` and as long as you then implement an `Iterator` subtype for your data abstraction, the abstraction can be used as the target in a for-each loop.

Iterator Design Pattern

The Iterator design pattern, as presented in the GoF Design Patterns book [2, p.257], provides "a way to access the elements of an aggregate object sequentially without exposing its underlying representation". Figure 1 presents a simplified version of the structure presented for the Iterator design pattern. Can you see the similarity to the discussion above of iteration abstraction in Java? (In the figure, each class or interface is represented as a box with operations listed below the name of the class or interface.)

Here is an on-line version of Iterator design pattern (http://www.vincehuston.org/dp/iterator.html).

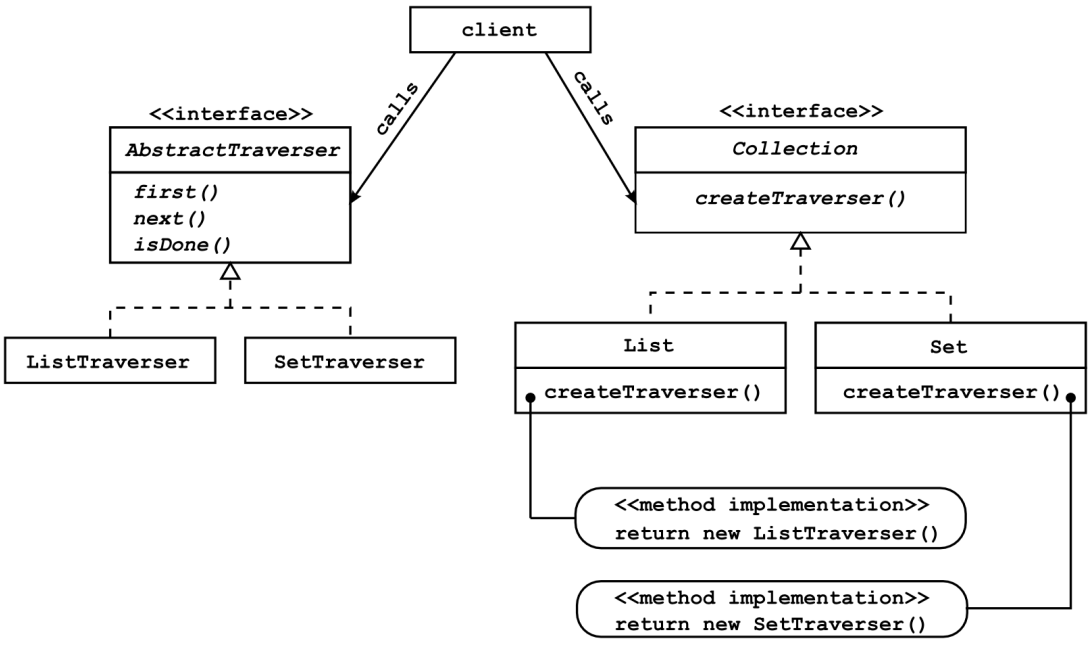

Figure 1: The Iterator Design Pattern (simplified)

References and Further Reading

Material in this reading is based on:

[1] Program Development in Java: Abstraction, Specification and Object-oriented Design by Barbara Liskov with John Guttag. Addison-Wesley, 2001.

Liskov is the 2008 winner of the ACM A.M. Turing Award which is the most prestigious award in computer science. She was awarded this honour for foundational innovations in programming language design.

This book provides a more comprehensive description of abstraction mechanisms and the kinds of abstractions if you are interested in gaining a deeper understanding of this material.

[2] Design Patterns: Elements of Reusable Object-Oriented Software by Gamma, Helm, Johnson and Vlissides. Addison-Wesley.