

Hands-On: Iterators

Iterators - Purpose and Overview

The *iterator pattern* is one of the most common programming patterns that we use. It describes the standard solution to the common problem of accessing every element of a collection in turn. We can think of the iterator pattern as *linear scan* over a collection. Java provides the `Iterator` interface as the standard means of expressing iteration. There are actually two interfaces relevant to iteration: `Iterator` and `Iterable`. You can read more about each at the following links.

- <https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/Iterator.html>
- <https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/lang/Iterable.html>

The real power of an iterator is this: Through the `Iterator` interface, we can traverse a collection of elements **without knowing exactly what the collection is or how it is structured**. Now we can perform a linear scan on ... *anything*.

For us to iterate over something, that something will have to provide us with an `Iterator` object over its elements. The standard (idiomatic) way of doing that in Java is for the “something” to implement the `Iterable` interface, thereby obliging it to provide a method named `iterator` that returns an `Iterator` object over its elements. Let’s say that this “something” is a class named `AnIterableCollection`. Then the shell of the class must look something like this:

```
public class AnIterableCollection<T> implements Iterable<T> {  
    // fields, methods, etc.  
    public Iterator<T> iterator() {  
        // create an Iterator object over the elements in this collection.  
    }  
}
```

Many of the collection classes in the [Java Collections Framework \(JCF\)](#) are *iterable* so we can use `ArrayList` for an example. Suppose we have a class named `Song` that models music that we listen to, and suppose we use an `ArrayList` to create a playlist of songs for game day.

```
ArrayList<Song> playlist = new ArrayList<>();  
playlist.add(new Song("War Eagle"));  
playlist.add(new Song("Welcome to the Jungle"));  
playlist.add(new Song("All I Do Is Win"));  
playlist.add(new Song("Sweet Caroline"));
```

Since the `ArrayList` class implements the `Iterable` interface, we can use standard iteration patterns to play each song in our playlist. The `for-each` loop is the standard idiom in Java for

iterating over `Iterable` objects.

```
for (Song song : playlist) {  
    song.play();  
}
```

The `for-each` loop is just a more compact form of the following more general pattern:



```
Iterator<Song> itr = playlist.iterator();  
while (itr.hasNext()) {  
    itr.next().play();  
}
```



For now we will mostly use one of these two patterns, but it's worth pointing out that Java offers a feature called [lambda expressions](#) which provide us with a very compact iteration idiom.

```
playlist.forEach(song -> song.play());
```

We will stick to the traditional patterns for this course, but features such as lambda expressions and [streams](#) may be something you'd like to explore on your own.

Iterators - an example

1. Open [IteratorExample.java](#) in jGRASP and study the source code.
 - Notice the use of the iterator pattern in the `toString` method.
 - Notice that the parameter to `toString` is `Iterable`. This allows an instance of any class that implements the `Iterable` interface to be passed in.
 - Notice how the `main` method passes in four very different collections, but the iterator pattern abstracts away the differences. The `toString` method doesn't care what the specific collection actually is; it only cares that whatever collection is passed in is guaranteed to be `Iterable`.
2. Run `IteratorExample.java` and observe its output.
3. Use the provided jGRASP Canvas file ([IteratorExample.jgrasp_canvas.xml](#)) to watch the execution of this code. Once you save the jGRASP Canvas XML file to the same directory as the source code, you can use the jGRASP Canvas like so:
 - Click on the Canvas file in the jGRASP Browse tab or click on the *Run in Canvas* button  in the jGRASP tool bar.
 - Once the Canvas window opens, resize it so that it best fits your display.
 - You can use any of the following controls to watch the program execute.
 - *Play*  - Starts the program running in auto-step mode.

- *Step Over*  - Manually steps over each statement.
- *Step In*  - Manually steps into each statement (method call).

4. Study the source code again and make sure you understand how the `Iterator` interface allows a single method to traverse four distinct collection types.
5. Modify the body of the `toString` method so that it uses the `for-each` iterator idiom instead.

Iterators - a common error

A common error in using iterators arises from a misunderstanding of what the `next()` method actually does. You can identify and learn to correct this common error through the steps below.

1. Open [IteratorError.java](#) in jGRASP and compile it.
2. Run `IteratorError.java` and observe its output.
3. Identify and correct the errors in use of the iterator. Hint: You may decide use a different expression for the iteration altogether to avoid the error.
4. It might help you identify the errors if you use the provided jGRASP Canvas file ([IteratorError.jgrasp_canvas.xml](#)) in conjunction with the jGRASP debugger.

Submission

The submission page for this activity asks you to submit your corrected `IteratorError.java` a grade.