P7 ITERATING TO PHILOSOPHY

Posted on March 8, 2019 by dahl

- **SUBMIT your completed assignment by 9:59PM on Wednesday, March 27th 2019.** We will accept and grade submissions made through 9:59PM on Thursday, March 28th 2019 (HARD DEADLINE). But, NO CREDIT will be granted for any work that is submitted even one second after this hard deadline.
- **Pair Programming is NOT allowed for this assignment. You have to work on and complete this assignment INDIVIDUALLY.**

This Assignment involves developing increasingly general Iterators using both the Iterator and Iterable interfaces from the Java API. Rather than iterating through a pre-existing collection of data, your code will be generating this data on demand. The final generalization of this pattern will be used to explore the structure of Wikipedia and the significance of its Philosophy page.

OBJECTIVES AND GRADING CRITERIA

The main goals of this assignment include giving you practice and experience with both Generics and Iterators in Java. You will implement the Iterable and Iterator interfaces of the Java API, and practice generalizing your code to be useful in a wider variety of applications.

20	Online Tests: these automated grading test results are visible upon uploading your submission. You are allowed multiple opportunities to correct the organization and functionality of your code (if necessary).
20 points	Offline Tests: these automated grading tests are run after the assignment's deadline has passed. They check for similar functionality and organizational correctness as the Online Tests. Since you will not have opportunities to make corrections after seeing the feedback from these tests, you should consider and test the correctness of your own code as thoroughly as possible.
10 points	Code Readability: human graders will review the commenting, style, and organization of your final submission. They will be checking whether it conforms to the requirements of the CS300 Course Style Guide. Since you will not have opportunities to make corrections after seeing the feedback from these graders, you should consider and review the readability of your own code as thoroughly as possible.

0. GETTING STARTED

Create a new Java8 project in Eclipse. You can name this project whatever you like, but P7 Iterating to Philosophy is a descriptive choice. Then create a new class named TestDriver with a public static void main(String[] args) method stub.

You'll be submitting your code for this assignment through gradescope. The six files that you will be submitting include: TestDriver.java, EvenNumbers.java, InfiniteIterator.java, FiniteIterator.java, Generator.java, and NextWikiLink.java.\

This assignment will be different from some past assignments in that you must decide what private instance fields (if any) are necessary to implement the following classes with the methods that behave as described below.

1. EVEN NUMBER ITERATOR

Create a new class named EvenNumbers that implements Iterator<Integer> and generates a sequence of even numbers started a specified starting point. The constructor for this class should take a single Integer parameter as input, and this Integer should be returned from that EvenNumber object's next() method on the first time that it is called. Each subsequent call of the next() method should return the smallest even number that is larger than the previously returned one. This should continue for as many times as the next() method may be called, which means that the hasNext() method for this class should always return true.

Assume that only even numbers are passed into the constructor for this class, and that the next() method will never be called so many times that an value larger than what can be held in a single primitive int will be returned. You do not need to write any special code to handle such situations.

Add the following method to your TestDriver class, and make sure that it returns true before moving on.

```java
public static boolean testEvenNumbers() {
  EvenNumbers it = new EvenNumbers(44);
  if(it.next() != 44) {
    System.out.println("The first call of EvenNumbers.next() "
        + "did not return the value passed into its constructor.");
    return false;
  }
  if(it.next() != 46) {
    System.out.println("The second call of EvenNumbers.next() "
        + "did not return the smallest even number, "
        + "that is larger than the previously returned number.");
    return false;
  }
  if(it.hasNext() != true) {
    System.out.println("EvenNumbers.next() returned false, "
        + "but should always return true.");
    return false;
  }
  return true;
}
```

2. INFINITE ITERATOR (POWERS OF TWO)

Make a copy of your EvenNumbers class, but rename this copy InfiniteIterator. The code in this class already generates an infinite sequence of Integer values. But we'd like to make this class even more general. Instead of only returning sequences of even numbers, we'd like it to generate any sequence in which the next term in the sequence can be calculated based on the previous term. For example, we might want to generate powers of two. Each term in this sequence will simply be the previous term multiplied by two. First we need a place to put this code. Since this is such a small helper-class, we'll put it's definition inside the TestDriver.java source file (instead of within a separate file of its own). At the bottom of this file (after the end of the TestDriver class definition) add the following class definition.

```java
class NextPowerOfTwo implements Function<Integer,Integer> {
  @Override
  public Integer apply(Integer previous) {
    return 2*previous;
  }
}
```

Notice that the Function<T,R> interface only requires that we implement a single method apply(), which takes and argument of type T and returns a value of type R. This makes it a great general type for us to use when generalizing our InfiniteIterator. Update your InfiniteIterator's constructor to take a second parameter of type Function<Integer,Integer>, and then use that object to generate the next value that this iterator returns. Note that you do not need to define any new generic type parameters for any classes or methods in this step (this will be added in the next step), although you clearly need to specify the generic type arguments related to each use of this Function interface. The following test can be used to check that this interface is designed correctly. Add this test to the body of your TestDriver class.

```java
public static boolean testPowersOfTwo() {
  InfiniteIterator it = new InfiniteIterator(8, new NextPowerOfTwo());
  if(it.next() != 8) {
    System.out.println("The first call of InfiniteIterator.next() "
        + "did not return the number passed into its constructor.");
    return false;
  }
  if(it.next() != 16) {
    System.out.println("The second call of InfiniteIterator.next() "
        + "did not return the smallest power of two number, "
        + "that is larger than the previously returned number.");
    return false;
  }
  if(it.hasNext() != true) {
    System.out.println("InfiniteIterator.next() returned false, "
        + "but should always return true.");
    return false;
  }
  return true;
}
```

3. INFINITE ITERATOR (EXTRA SMILES)

Our infinite generator can make use of any method that generates a next integer based on the last integer. Let's now further generalize this class, so that it can work with any type of data (not just Strings). We'll still wrap this function for generating the next value based on the previous into a class like we did with NextPowerOfTwo. Here's an example that adds an extra smile face to the end of a string 🙂

```java
class AddExtraSmile implements Function<String,String> {
  @Override
  public String apply(String t) {
    return t + " :)";
  }
}
```

Update your InfiniteIterator class so that it works with Strings, Integers, and other types of objects, as long as it receives a Function object for calculating the next base on the previous. You should define a new Generic type parameter to accomplish this. Here's a test that you can run with your more general InfiniteIterator. Note that the old tests should also continue to work after adding the generic type argument Integer to your definition of the InfiniteIterator variable "it".

```java
public static boolean testAddExtraSmile() {
  InfiniteIterator<String> it = new InfiniteIterator<>("Hello", new AddExtraSmile());
  if(!it.next().equals("Hello")) {
    System.out.println("The first call of InfiniteIterator.next() "
        + "did not return the string passed into its constructor.");
    return false;
  }
  if(!it.next().contains(" :)")) {
    System.out.println("The second call of InfiniteIterator.next() "
        + "did not return the a string with one more :), "
        + "than the previously returned string.");
    return false;
  }
  if(it.hasNext() != true) {
    System.out.println("InfiniteIterator.next() returned false, "
        + "but should always return true.");
    return false;
  }
  return true;
}
```

4. FINITE ITERATOR

So far, all of our iterators' hasNext() methods will only ever return true. This ability to represent sequences that are infinite can be powerful, but there are also many application in which only a finite number of elements will be needed. Let's create a new generic FiniteIterator class (which implements the Iterator interface) to retrieve a fixed number of elements from any InfiniteIterator. The constructor for this new class must take a generic InfiniteIterator, and an int length as input its two parameters (in that order). It's next method should return the value returned from a call to the provided InfiniteIterator's next() method. After the first "length" calls of the FiniteIterator's next() method have been made, it's hasNext() method should begin returning false instead of true. Here is a test method to help demonstrate how your FiniteIterator should be used:

```
public static boolean testFiniteIterator() {
  InfiniteIterator<Integer> infinite = new InfiniteIterator<>(2, new NextPowerOfTwo());
  FiniteIterator<Integer> it = new FiniteIterator<>(infinite, 8);
  String s = "";
  while(it.hasNext())
    s += " " + it.next();
  if(!s.equals(" 2 4 8 16 32 64 128 256")) {
    System.out.println("Repeatedly called the next() method of a FiniteIterator,"
        + "and the incorrect valuese were returned:" + s);
    return false;
  }
  return true;
}
```

5. GENERATOR ITERABLE

Now that we have some nice general iterators that are capable of generating finite and infinite sequences of values, let's expose the ability to create such iterators through a new class (with a generic type parameter) that implements the interface: Iterable. We'll name this new class Generator, and will define the following two overloaded constructors for this class:

```
public Generator(T firstValue, Function<T,T> generateNextFromLast) {}
public Generator(T firstValue, Function<T,T> generateNextFromLast, int length) {}
```

The iterator() method called on object constructed with the first of these constructors, should return a new InfiniteIterator that makes use of the provided firstValue and generateNextFromLast. When the second constructor is used to construct a new Generator, it's iterator() method will instead create and return a new FiniteIterator using the specified length. After you have implemented this Generator class, create your own test method with the following signature inside your TestDriver class. This test method should make use of Java's for-each loop to step through the contents of a finite sequence of your choice that is created by Generator. If you'd like to create a new Function object for this test, please add it to the bottom of the TestDriver.java source file (beneath the definitions of NextPowerOfTwo, and AddExtraSmile).

```java
public static boolean testGenerator() {}
```

6. PHILOSOPHY AND SCIENCE

Now that we have developed a general Generator class, let's make use of that Generator to implement one final application. An interesting observation was made several years back, that the following process will lead you to the Wikipedia page for Philosophy, when followed from as many as 97% or random starting Wikipedia pages. 1) Choose a random starting Wikipedia page, 2) follow the first link within this page's description text (excluding pronunciation and citation links), and then 3) repeat step 2 until the page for Philosophy has been reached. Here's a short video about this observation: https://www.youtube.com/watch?v=Q2DdmEBXTpo. Note that since Wikipedia pages can be edited by anyone, some of the more common paths to Philosophy are occasionally been changed, and that "Science" seems to be another common landing page that results from this process. Let's make our own application to explore these patterns further.

This process of following the first link in one Wikipedia page to the next, could be modeled as a kind of Iterator that is created by our Generator class. We just need to implement a method that can follow the link in one Wikipedia page to the next, within an object that implements the Function interface. Here is some code for such an implementation. In order to extract data from these web pages, it makes use of the jsoup library, which you'll need to download from the .jar file here. Adding this .jar file to your project, and right-click on it in the Project Explorer to add it to the build path of your project. After doing this, add a new class named NextWikiLink to your project, and fill it with the following code.

```java
import java.io.IOException;
import java.util.function.Function;
import org.jsoup.Jsoup;
import org.jsoup.nodes.Document;
import org.jsoup.select.Elements;

public class NextWikiLink implements Function<String, String> {
  @Override
  public String apply(String t) {
    try {
      // Download a Wikipedia page, using t in their internal link format: /wiki/Some_Subject
      Document doc = Jsoup.connect("https://en.wikipedia.org" + t).get();
      // Use .css selector to retrieve a collection of links from this page's description
      // "p a" selects links within paragraphs
      // ":not([title^='Help'])" skips pronunciations
      // ":not(sup a)" skips citations
      Elements links = doc.select("p a:not([title^='Help']):not(sup a)");
      // return the link attribute from the first element of this list
      return links.get(0).attr("href");
      // Otherwise return an appropriate error message:
    } catch (IOException | IllegalArgumentException e) {
      return "FAILED to find wikipedia page: " + t;
    } catch (IndexOutOfBoundsException e) {
      return "FAILED to find a link in wikipedia page: " + t;
    }
  }

  public static void main(String[] args) {
    // Implement your own Wikipedia crawling application here.
    // 1. prompt the user to enter a topic name and number of iterations to follow
    // 2. prepend "/wiki/" to the user's input, and replace spaces with underscores
    // 3. use a for-each loop to iterate through the number of links requested
  }
}
```

Your last task for this assignment is to implement the main method within this class. Note that if you also have a main method in your TestDriver class, there will be two within this project. You can always right-click the source file containing the main() method that you would like to run, and choose to Run As... Java Application from there, in case Eclipse gets confused about which main() method you intend to execute. The commented algorithm for this main method is quite simple, but here are a couple of sample executions of this program (note that the specific pages linked to may change, even while this assignment is being completed).

```
Enter a wikipedia page topic: computer programming
Enter the number of pages you'd like to step through: 12
/wiki/computer_programming
/wiki/Executable
/wiki/Computing
/wiki/Computer
/wiki/Sequence
/wiki/Mathematics
/wiki/Ancient_Greek
/wiki/Greek_language
/wiki/Modern_Greek
/wiki/Colloquialism
/wiki/Linguistics
/wiki/Science
```

```
Enter a wikipedia page topic: no such page
Enter the number of pages you'd like to step through: 12
FAILED to find wikipedia page: /wiki/no_such_page
```

(Update 3/24) It is acceptable for your program to print out the /wiki/no_such_page line before displaying the Failed to find wikipedia page: /wiki/no_such_page.

SUBMISSION

Congratulations on finishing this CS300 assignment! After verifying that your work is correct, and written clearly in a style that is consistent with the course style guide, you should submit your final work through gradescope.com. The six source files that you must submit include: TestDriver.java, EvenNumbers.java, InfiniteIterator.java, FiniteIterator.java, Generator.java, and NextWikiLink.java. Your score for this assignment will be based on your "active" submission made prior to the hard deadline of **9:59PM on Thursday, March 28th**.

EXTRA CHALLENGES

Here are some suggestions for interesting ways to extend this simulation, after you have completed, backed up, and submitted the graded portion of this assignment. **No extra credit will be awarded for implementing these features**, but they should provide you with some valuable practice and experience.

- Try creating new Iterators that take other Iterators as input. For example, you could create an iterator that takes two others as input to its constructor, and then alternates between returning the next value produced by one and then by the other. This kind of computation is often called a zip.
- Another kind of Iterator that you can create, is one that takes another iterator as input and but only returns some of the values produced by that iterator. This kind of computation is often called a filter. Note that the filter criteria could be encoded within a Function<T,Boolean>.
- Find other interesting applications for the Iterators and "Generator" that you have created in this assignment. For example, could you create a Particle[] Generator, that allow you to update all of the particles within that array by calling next()? This could be used to refactor your code from P2 ParticleFountain?