CS2030S Mini Project: Infinite List

Tags & Categories

Tags:

Categories:

Related Tutorials

Task Content

Infinite List

Topic Coverage

- Stream concepts
- Lambda functions
- Variable capture and closures
- Immutability

Problem Description

In the lectures, we have implemented an immutable list that is finite. Now, you are going to implement an infinite version, which is recursive and makes heavy use of the laziness of lambda expressions. Here are some essential properties on how our infinite list should behave:

- An infinite list needs to be as lazy as possible and only generate the elements from the data source when necessary
- An infinite list pipeline needs to be immutable
- When an element is generated, it should not be generated again. We do this by caching a copy of the value the first time an element is generated. Subsequent probing of the same value will result in the cached copy being returned. *You will see the need for this once we go stateful.*

The Task

You are to design an implementation of an `InfiniteList<T>` that supports the following operations:

Data sources:

- `public static <T> InfiniteList<T> generate(Supplier<? extends T> s)`
- `public static <T> InfiniteList<T> iterate(T seed, UnaryOperator<T> next)`

Terminal operations:

- `public long count();`
- `public <U> U reduce(U identity, BiFunction<U, ? super T, U> accumulator);`
- `public Object[] toArray();`
- `public void forEach(Consumer<? super T> action);`

Stateless intermediate operations:

- `public InfiniteList<T> filter(Predicate<? super T> predicate);`
- `public <R> InfiniteList<R> map(Function<? super T, ? extends R> mapper);`

Stateful intermediate operations:

- `public InfiniteList<T> limit(long n);`
- `public InfiniteList<T> takeWhile(Predicate<? super T> predicate);`

Since `InfiniteList` is similar to Java's `Stream`, you are **not allowed** to import packages from `java.util.stream`.

The `InfiniteList` **interface** has also been provided for you with most of the methods commented out. It can be downloaded here. You can uncomment individual methods as you proceed through the levels. Note that an uncommented version of this same interface will be used when testing in CodeCrunch.

This task is divided into several levels. Read through all the levels to see how the different levels are related.

Remember to:

- always compile your program files first before either using `jshell` to test your program, or using `java` to run your program
- run `checkstyle` on your code

Level 1: Getting Lazy

We will start by developing a building block for our `InfiniteList`: an abstraction for an object of which the value is not computed until it is needed. Let's build a standalone generic class `Lazy<T>` (an extension of the one that was discussed in lecture) that encapsulates a value of type `T`, with the following operations and properties:

- `static <T> Lazy<T> ofNullable(T v)` creates a lazy object with a given value (precomputed) `v`. Note that the `Lazy` object can contain a `null` value.
- `static <T> Lazy<T> of(Supplier<? extends T> supplier)` creates a lazy object with a supplier to produce a value.
- `Optional<T> get()` returns an `Optional` of the contained value if it is not `null`; returns `Optional.empty` otherwise. If necessary (i.e. when the value is not already cached), `get()` triggers the computation of the value encapsulated in the `Lazy` instance (invoking the chain of supplier, predicates, and functions).
- `<R> Lazy<R> map(Function<? super T, ? extends R> mapper)` applies the function `mapper` lazily (i.e., only when needed) to the content of this calling `Lazy` instance and returns the new `Lazy` instance. If, upon evaluation, the content is `null`, then a lazy object containing `null` is returned.
- `Lazy<T> filter(Predicate<? super T> predicate)` applies the given predicate lazily to test the content of the calling `Lazy` instance. If the predicate returns `true`, the content is retained, otherwise, the content becomes `null`. If the content is already `null`, then the calling object is returned.
- `String toString()` returns "`Lazy[?]`" if the value has not been computed; the string representation of the value enclosed in `Lazy[..]` otherwise.

As `Lazy` is the context that handles `null` values and caching

- You are allowed to use the `null` value in the `Lazy` class,
- Object properties need not be declared `final`; however they should still be `private`.

Implementing `Lazy` and understanding how `Lazy` works is going to be very useful in later levels.

```
jshell> /open Lazy.java
jshell> Lazy.ofNullable(4)
$.. ==> Lazy[4]
jshell> Lazy.ofNullable(4).get()
$.. ==> Optional[4]
jshell> Lazy.ofNullable(4).map(x -> x + 4)
$.. ==> Lazy[?]
jshell> Lazy.ofNullable(4).filter(x -> x > 2)
$.. ==> Lazy[?]
jshell> Lazy.ofNullable(4).map(x -> 1).get()
$.. ==> Optional[1]
jshell> Lazy.ofNullable(4).filter(x -> true).get()
$.. ==> Optional[4]
jshell> Lazy.ofNullable(4).filter(x -> false).get()
$.. ==> Optional.empty
jshell> Lazy.ofNullable(4).map(x -> 1).filter(x -> false).get()
$.. ==> Optional.empty
jshell> Lazy.ofNullable(4).filter(x -> false).map(x -> 1).get()
$.. ==> Optional.empty
jshell> Lazy.ofNullable(4).filter(x -> true).map(x -> 1).get()
$.. ==> Optional[1]
jshell> Lazy.ofNullable(4).filter(x -> false).filter(x -> true).get()
$.. ==> Optional.empty
jshell>
jshell> Lazy.ofNullable(null)
$.. ==> Lazy[null]
jshell> Lazy.ofNullable(null).get()
$.. ==> Optional.empty
```

```
jshell> Lazy.ofNullable(null).map(x -> 1)
$.. ==> Lazy[?]
jshell> Lazy.ofNullable(null).filter(x -> true)
$.. ==> Lazy[?]
jshell> Lazy.ofNullable(null).filter(x -> false)
$.. ==> Lazy[?]
jshell> Lazy.ofNullable(null).map(x -> 1).get()
$.. ==> Optional.empty
jshell> Lazy.ofNullable(null).filter(x -> true).get()
$.. ==> Optional.empty
jshell> Lazy.ofNullable(null).filter(x -> false).get()
$.. ==> Optional.empty
jshell>
jshell> Lazy.of(() -> 4)
$.. ==> Lazy[?]
jshell> Lazy.of(() -> 4).get()
$.. ==> Optional[4]
jshell> Lazy.of(() -> 4).map(x -> 1)
$.. ==> Lazy[?]
jshell> Lazy.of(() -> 4).filter(x -> true)
$.. ==> Lazy[?]
jshell> Lazy.of(() -> 4).map(x -> 1).get()
$.. ==> Optional[1]
jshell> Lazy.of(() -> 4).filter(x -> true).get()
$.. ==> Optional[4]
jshell> Lazy.of(() -> 4).filter(x -> false).get()
$.. ==> Optional.empty
jshell> Lazy<Integer> lazy = Lazy.of(() -> 4)
jshell> lazy
lazy ==> Lazy[?]
jshell> lazy.get()
$.. ==> Optional[4]
jshell> lazy
lazy ==> Lazy[4]
jshell>
jshell> Lazy.of(() -> null)
$.. ==> Lazy[?]
jshell> Lazy.of(() -> null).get()
$.. ==> Optional.empty
jshell> Lazy.of(() -> null).map(x -> 1)
$.. ==> Lazy[?]
jshell> Lazy.of(() -> null).filter(x -> false)
$.. ==> Lazy[?]
jshell> Lazy.of(() -> null).map(x -> 1).get()
$.. ==> Optional.empty
jshell> Lazy.of(() -> null).filter(x -> true).get()
$.. ==> Optional.empty
jshell> Lazy.of(() -> null).filter(x -> false).get()
$.. ==> Optional.empty
jshell> Lazy<Integer> lazy = Lazy.of(() -> null)
jshell> lazy
lazy ==> Lazy[?]
jshell> lazy.get()
$.. ==> Optional.empty
jshell> lazy
lazy ==> Lazy[null]
```

Compile your program by running the following on the command line:

```
$ javac -Xlint:rawtypes *.java
```

Level 2: To Infinity and Beyond

Define the `InfiniteListImpl` class that implements the `InfiniteList` interface. Write the static `generate` and `iterate` methods that initiate the initial list pipeline. You are encouraged to adapt the implementation demonstrated in class using the head and tail suppliers. If you do it right, you need only have these two declared `private final` and nothing else. That said, if you feel compelled to introduce other fields to aid in your implementation, feel free to do so. But note that the implementation demonstrated in class does not support caching of computed values. To solve this lab, you will need to use the type `Lazy<T>` for the head to take advantage of its laziness, instead of `Supplier<T>`.

For debugging purposes, include a method `InfiniteList<T> peek()` within your implementation that prints the first element of the infinite list to the standard output and returns the rest of the list as an `InfiniteList`. This enables the chaining of `peek()` methods to produce an output of a sequence of stream elements. Note that `peek()` is solely for the purpose of debugging, until a terminal operation is introduced at a later level.

```
jshell> InfiniteList<Integer> list
jshell> InfiniteList.generate(() -> 1) instanceof InfiniteListImpl
$.. ==> true
jshell> InfiniteList.iterate(1, x -> x + 1) instanceof InfiniteListImpl
$.. ==> true
jshell> list = InfiniteListImpl.generate(() -> 1).peek()
1
jshell> list = InfiniteListImpl.iterate(1, x -> x + 1).peek()
1
jshell> list = InfiniteListImpl.iterate(1, x -> x + 1).peek().peek()
1
2
jshell> InfiniteList<Integer> list2 = list.peek()
3
jshell> list != list2
$.. ==> true
jshell> InfiniteList<String> list = InfiniteListImpl.iterate("A", x -> x + "Z").peek().pe
A
AZ
AZZ
jshell>
jshell> UnaryOperator<Integer> op = x -> { System.out.printf("iterate: %d -> %d\n", x, x
jshell> list2 = InfiniteList.iterate(1, op).peek().peek()
1
iterate: 1 -> 2
2
iterate: 2 -> 3
jshell>
jshell> /exit
```

Compile your program by running the following on the command line:

```
$ javac -Xlint:rawtypes *.java
$ jshell -q Lazy.java InfiniteList.java InfiniteListImpl.java < level2.jsh
```

Check your styling by issuing the following

```
$ checkstyle *.java
```

Level 3: Map and Filter

Now implement the `map` and `filter` operations. In particular, if an element from an upstream operation does not pass through `filter`, an `Optional.empty()` will be generated. `peek()` does not print anything if the element is `Optional.empty()`.

```
jshell> InfiniteList<Integer> list, list2
jshell> list = InfiniteList.generate(() -> 1).map(x -> x * 2)
jshell> list2 = list.peek()
2
jshell> list2 = list.peek()
2
jshell> InfiniteList.generate(() -> 1).map(x -> x * 2) instanceof InfiniteListImpl
$.. ==> true
jshell> list = InfiniteList.generate(() -> 1).map(x -> x * 2).peek()
2
jshell> list = InfiniteList.generate(() -> 1).map(x -> x * 2).peek().peek()
2
2
jshell> list = InfiniteList.iterate(1, x -> x + 1).map(x -> x * 2).peek().peek()
2
4
jshell>
jshell> Supplier<Integer> generator = () -> { System.out.println("generate: 1"); return 1
jshell> Function<Integer,Integer> doubler = x -> { System.out.printf("map: %d -> %d\n", x
jshell> Function<Integer,Integer> oneLess = x -> { System.out.printf("map: %d -> %d\n", x
jshell> list = InfiniteList.generate(generator).map(doubler).peek().peek()
generate: 1
map: 1 -> 2
2
```

```
generate: 1
map: 1 -> 2
2
jshell> list = InfiniteList.generate(generator).map(doubler).map(oneLess).peek().peek()
generate: 1
map: 1 -> 2
map: 2 -> 1
1
generate: 1
map: 1 -> 2
map: 2 -> 1
1
jshell>
jshell> list = InfiniteList.iterate(1, x -> x + 1).filter(x -> x % 2 == 0)
jshell> list2 = list.peek()
jshell> list2 = list.peek()
jshell> InfiniteList.iterate(1, x -> x + 1).filter(x -> x % 2 == 0) instanceof InfiniteLi
$.. ==> true
jshell> list = InfiniteList.iterate(1, x -> x + 1).filter(x -> x % 2 == 0).peek().peek()
2
jshell> list = InfiniteList.iterate(1, x -> x + 1).filter(x -> x % 2 == 0).filter(x -> x
2
jshell>
jshell> Predicate<Integer> isEven = x -> { System.out.printf("filter: %d -> %b\n", x, x %
jshell> Predicate<Integer> lessThan10 = x -> { System.out.printf("filter: %d -> %b\n", x,
jshell> UnaryOperator<Integer> op = x -> { System.out.printf("iterate: %d -> %d\n", x, x
jshell> list = InfiniteList.iterate(1, op).filter(isEven).peek().peek()
filter: 1 -> false
iterate: 1 -> 2
filter: 2 -> true
2
iterate: 2 -> 3
jshell> list = InfiniteList.iterate(1, op).filter(isEven).filter(lessThan10).peek().peek(
filter: 1 -> false
iterate: 1 -> 2
filter: 2 -> true
filter: 2 -> true
2
iterate: 2 -> 3
jshell>
jshell> list = InfiniteList.iterate(1, op).map(doubler).filter(isEven).filter(lessThan10)
map: 1 -> 2
filter: 2 -> true
filter: 2 -> true
2
iterate: 1 -> 2
map: 2 -> 4
filter: 4 -> true
filter: 4 -> true
4
iterate: 2 -> 3
jshell> list = InfiniteList.iterate(1, op).filter(isEven).map(doubler).filter(lessThan10)
filter: 1 -> false
iterate: 1 -> 2
filter: 2 -> true
map: 2 -> 4
filter: 4 -> true
4
iterate: 2 -> 3
jshell> list = InfiniteList.iterate(1, op).filter(isEven).filter(lessThan10).map(doubler)
filter: 1 -> false
iterate: 1 -> 2
filter: 2 -> true
filter: 2 -> true
map: 2 -> 4
4
iterate: 2 -> 3
jshell> /exit
```

Compile your program by running the following on the command line:

```
$ javac -Xlint:rawtypes *.java
$ jshell -q <your java files> < level3.jsh
```

Check your styling by issuing the following

```
$ checkstyle *.java
```

Level 4: Emptiness and Limitations

Now, create a subtype of `InfiniteList` that represents an empty list called `EmptyList`. The `EmptyList` should return `true` when `isEmpty()` is called and it should return itself for intermediate operations and appropriate values for terminal operations.

Then, implement the `limit` method. There is now a need to differentiate between an `Optional.empty()` produced from `filter` and the end of the stream in `limit`. Make use of `EmptyList` to indicate the end of the stream. When dealing with `limit`, you will need to decide if the upstream element

- produces an empty list;
- produces an `Optional.empty` and ignored by `limit`; or
- produces a stream element and accounted for by `limit`

The other operation to truncate an infinite list is the `takeWhile` operator. The same considerations that you have given to `limit` would probably apply here.

```
jshell> InfiniteList.iterate(1, x -> x + 1).isEmpty()
$.. ==> false
jshell> InfiniteList.generate(() -> 2).isEmpty()
$.. ==> false
jshell> InfiniteList.generate(() -> 2).filter(x -> x % 3 == 0).isEmpty()
$.. ==> false
jshell> InfiniteList.iterate(1, x -> x + 1).map(x -> 2).isEmpty()
$.. ==> false
jshell> InfiniteList.iterate(1, x -> x + 1).filter(x -> x % 2 == 0).isEmpty()
$.. ==> false
jshell> new EmptyList<>().isEmpty()
$.. ==> true
jshell> new EmptyList<>().map(x -> 2).isEmpty()
$.. ==> true
jshell> new EmptyList<>().filter(x -> true).isEmpty()
$.. ==> true
jshell> new EmptyList<>().filter(x -> false).isEmpty()
$.. ==> true
jshell>
jshell> InfiniteList.iterate(1, x -> x + 1).limit(0).isEmpty()
$.. ==> true
jshell> InfiniteList.iterate(1, x -> x + 1).limit(1).isEmpty()
$.. ==> false
jshell> InfiniteList.iterate(1, x -> x + 1).limit(-1).isEmpty()
$.. ==> true
jshell>
jshell> UnaryOperator<Integer> op = x -> { System.out.printf("iterate: %d -> %d\n", x, x
jshell> InfiniteList.iterate(1, op).limit(0).isEmpty()
$.. ==> true
jshell> InfiniteList.iterate(1, op).limit(1).isEmpty()
$.. ==> false
jshell> InfiniteList.iterate(1, op).limit(2).isEmpty()
$.. ==> false
jshell> InfiniteList<Integer> list;
jshell> list = InfiniteList.iterate(1, op).limit(0).peek()
jshell> list = InfiniteList.iterate(1, op).limit(1).peek()
1
jshell> list = InfiniteList.iterate(1, op).limit(1).peek().peek()
1
jshell> list = InfiniteList.iterate(1, op).limit(2).peek().peek().peek()
1
iterate: 1 -> 2
2
jshell> list = InfiniteList.iterate(1, op).limit(2).limit(1).peek().peek()
1
jshell> list = InfiniteList.iterate(1, op).limit(1).limit(2).peek().peek()
1
jshell>
jshell> InfiniteList.iterate(1, op).takeWhile(x -> x < 0).isEmpty()
$.. ==> false
jshell> InfiniteList.iterate(1, op).takeWhile(x -> x < 2).isEmpty()
$.. ==> false
jshell> list = InfiniteList.iterate(1, op).takeWhile(x -> x < 0).peek()
jshell> list = InfiniteList.iterate(1, op).takeWhile(x -> x < 2).peek().peek()
1
iterate: 1 -> 2
jshell> list = InfiniteList.iterate(1, op).takeWhile(x -> x < 2).takeWhile(x -> x < 0).pe
jshell> list = InfiniteList.iterate(1, op).takeWhile(x -> x < 0).takeWhile(x -> x < 2).pe
jshell> list = InfiniteList.iterate(1, op).takeWhile(x -> x < 5).takeWhile(x -> x < 2).pe
1
iterate: 1 -> 2
jshell>
jshell> Predicate<Integer> lessThan5 = x -> { System.out.printf("takeWhile: %d -> %b\n",
jshell> list = InfiniteList.iterate(1, op).takeWhile(lessThan5).peek().peek()
takeWhile: 1 -> true
1
iterate: 1 -> 2
takeWhile: 2 -> true
2
iterate: 2 -> 3
jshell>
jshell> /exit
```

Compile your program by running the following on the command line:

```
$ javac -Xlint:rawtypes *.java
$ jshell -q Lazy.java InfiniteList.java InfiniteListImpl.java < level4.jsh
```

Check your styling by issuing the following

```
$ checkstyle *.java
```

Level 5: Terminals

Now, we are going to implement the terminal operations: `forEach`, `count`, `reduce`, and `toArray`.

```
jshell> new EmptyList<>().toArray()
$.. ==> Object[0] {  }
jshell> InfiniteList.iterate(0, i -> i + 1).limit(10).limit(3).toArray()
$.. ==> Object[3] { 0, 1, 2 }
jshell> InfiniteList.iterate(0, i -> i + 1).limit(3).limit(100).toArray()
$.. ==> Object[3] { 0, 1, 2 }
jshell> InfiniteList.generate(() -> 1).limit(0).toArray()
$.. ==> Object[0] {  }
jshell> InfiniteList.generate(() -> 1).limit(2).toArray()
$.. ==> Object[2] { 1, 1 }
jshell> Random rnd = new Random(1)
jshell> InfiniteList.generate(() -> rnd.nextInt() % 100).limit(4).toArray();
$.. ==> Object[4] { -25, 76, 95, 26 }
jshell> InfiniteList.generate(() -> "A").map(x -> x + "-").map(str -> str.length()).limit
$.. ==> Object[4] { 2, 2, 2, 2 }
jshell> InfiniteList.generate(() -> "A").limit(4).map(x -> x + "-").map(str -> str.length
$.. ==> Object[4] { 2, 2, 2, 2 }
jshell> InfiniteList.generate(() -> "A").map(x -> x + "-").limit(4).map(str -> str.length
$.. ==> Object[4] { 2, 2, 2, 2 }
jshell> InfiniteList.iterate(1, x -> x + 1).limit(4).filter(x -> x % 2 == 0).toArray()
$.. ==> Object[2] { 2, 4 }
jshell> InfiniteList.iterate(1, x -> x + 1).filter(x -> x % 2 == 0).limit(4).toArray()
$.. ==> Object[4] { 2, 4, 6, 8 }
jshell> InfiniteList.iterate(0, x -> x + 1).filter(x -> x > 10).filter(x -> x < 20).limit
$.. ==> Object[5] { 11, 12, 13, 14, 15 }
jshell> InfiniteList.iterate(0, x -> x + 1).filter(x -> x > 10).map(x -> x.hashCode() % 3
$.. ==> Object[5] { 11, 12, 13, 14, 15 }
jshell> InfiniteList.iterate(0, x -> x + 1).takeWhile(x -> x < 3).toArray()
$.. ==> Object[3] { 0, 1, 2 }
jshell> InfiniteList.iterate(0, x -> x + 1).takeWhile(x -> x < 0).toArray()
$.. ==> Object[0] {  }
jshell> InfiniteList.iterate(0, x -> x + 1).takeWhile(x -> x < 10).takeWhile(x -> x < 5).
$.. ==> Object[5] { 0, 1, 2, 3, 4 }
jshell> InfiniteList.iterate(0, x -> x + 1).map(x -> x).takeWhile(x -> x < 4).limit(1).to
$.. ==> Object[1] { 0 }
jshell> InfiniteList.iterate(0, x -> x + 1).limit(4).takeWhile(x -> x < 2).toArray()
$.. ==> Object[2] { 0, 1 }
jshell> InfiniteList.iterate(0, x -> x + 1).map(x -> x * x).filter(x -> x > 10).limit(1).
$.. ==> Object[1] { 16 }
```

```
jshell> InfiniteList.iterate(0, x -> x + 1).filter(x -> x > 10).map(x -> x * x).limit(1).
$.. ==> Object[1] { 121 }
jshell> Random rnd = new Random(1)
jshell> InfiniteList.generate(() -> rnd.nextInt() % 100).filter(x -> x > 10).limit(4).toA
$.. ==> Object[4] { 76, 95, 26, 69 }
jshell>
jshell> new EmptyList<>().count()
$.. ==> 0
jshell> InfiniteList.iterate(0, x -> x + 1).limit(0).count()
$.. ==> 0
jshell> InfiniteList.iterate(0, x -> x + 1).limit(1).count()
$.. ==> 1
jshell> InfiniteList.iterate(0, x -> x + 1).filter(x -> x % 2 == 1).limit(10).count()
$.. ==> 10
jshell> InfiniteList.iterate(0, x -> x + 1).limit(10).filter(x -> x % 2 == 1).count()
$.. ==> 5
jshell> InfiniteList.iterate(0, x -> x + 1).takeWhile(x -> x < 10).count()
$.. ==> 10
jshell> InfiniteList.iterate(0, x -> x + 1).takeWhile(x -> x < 10).filter(x -> x % 2 == 0
$.. ==> 5
jshell> Random rnd = new Random(1)
jshell> InfiniteList.generate(() -> Math.abs(rnd.nextInt()) % 100).takeWhile(x -> x > 5).
$.. ==> 9
jshell>
jshell> new EmptyList<Integer>().reduce(100, (x,y) -> x*y)
$.. ==> 100
jshell> InfiniteList.iterate(0, x -> x + 1).limit(5).reduce(0, (x, y) -> x + y)
$.. ==> 10
jshell> InfiniteList.iterate(0, x -> x + 1).limit(0).reduce(0, (x, y) -> x + y)
$.. ==> 0
jshell> InfiniteList.iterate(0, x -> x + 1).map(x -> x * x).limit(5).reduce(1, (x, y) -> 
$.. ==> 0
jshell> Random rnd = new Random(1)
jshell> InfiniteList.generate(() -> rnd.nextInt() % 100).filter(x -> x > 0).limit(10).red
$.. ==> 95
jshell>
jshell> UnaryOperator<Integer> op = x -> { System.out.printf("iterate: %d -> %d\n", x, x 
jshell> Supplier<Integer> generator = () -> { System.out.println("generate: 1"); return 1
jshell> Function<Integer,Integer> doubler = x -> { System.out.printf("map: %d -> %d\n", x
jshell> Function<Integer,Integer> oneLess = x -> { System.out.printf("map: %d -> %d\n", x
jshell> Predicate<Integer> lessThan100 = x -> { System.out.printf("takeWhile: %d -> %b\n"
jshell> Predicate<Integer> moreThan10 = x -> { System.out.printf("filter: %d -> %b\n", x,
jshell>
jshell> InfiniteList.iterate(0, op).filter(lessThan100).map(doubler).takeWhile(lessThan10
takeWhile: 0 -> true
map: 0 -> 0
takeWhile: 0 -> true
map: 0 -> -1
iterate: 0 -> 1
takeWhile: 1 -> true
map: 1 -> 2
takeWhile: 2 -> true
map: 2 -> 1
iterate: 1 -> 2
takeWhile: 2 -> true
map: 2 -> 4
takeWhile: 4 -> true
map: 4 -> 3
iterate: 2 -> 3
takeWhile: 3 -> true
map: 3 -> 6
takeWhile: 6 -> true
map: 6 -> 5
iterate: 3 -> 4
takeWhile: 4 -> true
map: 4 -> 8
takeWhile: 8 -> true
map: 8 -> 7
$.. ==> Object[5] { -1, 1, 3, 5, 7 }
jshell> InfiniteList.generate(generator).filter(lessThan100).map(doubler).takeWhile(lessT
generate: 1
takeWhile: 1 -> true
map: 1 -> 2
takeWhile: 2 -> true
```

```
map: 2 -> 1
generate: 1
takeWhile: 1 -> true
map: 1 -> 2
takeWhile: 2 -> true
map: 2 -> 1
generate: 1
takeWhile: 1 -> true
map: 1 -> 2
takeWhile: 2 -> true
map: 2 -> 1
generate: 1
takeWhile: 1 -> true
map: 1 -> 2
takeWhile: 2 -> true
map: 2 -> 1
generate: 1
takeWhile: 1 -> true
map: 1 -> 2
takeWhile: 2 -> true
map: 2 -> 1
$.. ==> Object[5] { 1, 1, 1, 1, 1 }
jshell>
jshell> new EmptyList<>().forEach(System.out::println)
jshell> InfiniteList.iterate(0, x -> x + 1).limit(0).forEach(System.out::println)
jshell> InfiniteList.iterate(0, x -> x + 1).limit(1).forEach(System.out::println)
0
jshell> InfiniteList.iterate(0, x -> x + 1).filter(x -> x % 2 == 1).limit(10).forEach(Sys
1
3
5
7
9
11
13
15
17
19
jshell> InfiniteList.iterate(0, x -> x + 1).limit(10).filter(x -> x % 2 == 1).forEach(Sys
1
3
5
7
9
jshell> InfiniteList.iterate(0, x -> x + 1).takeWhile(x -> x < 10).forEach(System.out::pr
0
1
2
3
4
5
6
7
8
9
jshell> InfiniteList.iterate(0, x -> x + 1).takeWhile(x -> x < 10).filter(x -> x % 2 == 0
0
2
4
6
8
jshell> /exit
```

Compile your program by running the following on the command line:

```
$ javac -Xlint:rawtypes *.java
```

You should try to test your implementation as exhaustively as you can before submitting to CodeCrunch. We shall be using another client class to test your implementation.