Week 4 - Model infinite sequences

Outcomes:

- Practice coding lazily and solving problems using laziness.
- Build more familiarity with class syntax.
- Method chaining as a way to make code clearer.
- Revise referential transparency and function purity.
- Functional Reactive Programming with Observables and Observers.

This week we're going to look at laziness. The best thing about lazy iterators, is that you can define infinite sequences with them, and only use what's needed. These techniques are especially useful when dealing with huge data sets in the real world (or other infinite sequences like user interaction with your program).

First you'll start by implementing an infinite sequence initialisation function (7.4 in the notes). This function will become very useful as you use it to approximate Pi (π=3.14159...), using techniques covered in the second week (closures) and third week (map, filter, reduce).

You'll then implement Observables which you can listen to with an Observer! This opens up a paradigm of programming called Functional Reactive Programming. An Observable allows a way to react to asynchronous data streams (keyboard presses, web-sockets, mouse moves, server requests).

You'll then use your observable to add visualisations to the approximation of π via Monte Carlo[3] method!

Exercise 1

Please implement a general purpose infinite sequence initialisation function! *Question 7.4 in the notes*. Make sure it passes the tests. It must also match the following type signature:

```
function initSequence<T>(transform: (value: T) => T): (initialValue: T) =>
LazySequence<T>
```

Note: `LazySequence<T>` is defined in your code and in the notes.

Exercise 2

Still following the notes, please implement map (question 7.3), filter (question 7.3), take(7.2) and reduce (question 7.3), making sure your functions match the given type signatures:

```
function map<T>(func: (v: T)=>T, seq: LazySequence<T>): LazySequence<T>
```

[3] A simple statistical method used to help invent the Atomic Bomb in Los Alamos, 1945.

```
function filter<T>(func: (v: T)=>boolean, seq: LazySequence<T>): LazySequence<T>
function take<T>(amount: number, seq: LazySequence<T>): LazySequence<T>
function reduce<T,V>(func: (v:V, t: T)=>V, seq: LazySequence<T>, start:V): V
```

Exercise 3 - Reduce everything!

Reduce is an extremely powerful tool for any algorithm that requires a single traversal of a structure. Therefore please implement the two unimplemented functions `maxNumber` and `lengthOfSequence`.

Use only reduce.

`maxNumber` should return the largest number in a lazy sequence.
`lengthOfSequence` should return the length of the lazy sequence.

If testing these yourself, make sure you wrap your lazy sequence in a `take` so that you don't get trapped in infinite recursion.

Exercise 4 - Lazy Pi approximations

Using the lazy list, let's approximate $\frac{\pi}{4}$. This can be done by defining an infinite series that looks like so:
$\frac{\pi}{4} = \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \ldots$
The series alternates between a plus and minus sign while the denominators are an ascending sequence of odd numbers starting from 1.

Write your solution into the function:

```
function exercise3Solution (seriesLength: number): number {
     // Your solution using lazy lists.
     // Use `take` to only take the right amount of the infinite list.
};
// Expect return of approximation of pi/4 based on the length of the series passed in.
```

This function should return whatever the approximation of $\frac{\pi}{4}$ is for `*seriesLength*` elements of the series.

Some ideas for solving this:

Technique 1 - Use reduce to accumulate the series.

Generate a sequence of odd numbers with alternating signs. (+1, -3, +5, -7, ...)
Use `take`, `reduce` and `map` to generate the approximation.

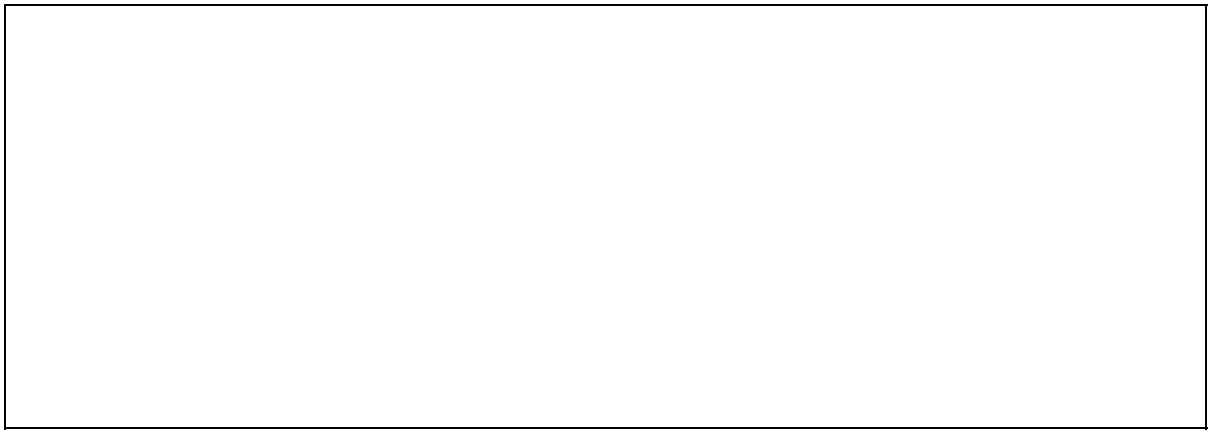
Technique 2 - Accumulating iterator

Another way is to create a generator that generates the approximation of $\frac{\pi}{4}$:

- Hint 1: Use closures like in Week 2's exercise 1.

- Hint 2: Does the order of arguments matter?

BIGGER HINT: A fibonacci sequence that only returns a single number and stores state (It's spoilers.) Highlight over it to see it:



Technique 3 - Whatever functional method you choose using an iterator.

Whatever you choose, you'll need to explain it to your demonstrator.

Exercise 5 - Listening to an Observable with an Observer, a naive and simple implementation

Provided is a lovely block of code demonstrating the simplest Observer interacting with the simplest Observable. The Observable in this case is a function that subscribes the Observer to the "data source". The "data source" is a timer that calls the Observer's `next` method with a random integer. After 1000 milliseconds the Observable calls the `complete` method on the Observer closing the Observable.

There is also an Observer called `loggingObserver`. Please answer the following questions in comments in-line in the code in the marked places. Each answer should only be a sentence or two:

a) What happens when you execute this code?
b) What is the side effect that occurs when `loggingObserver.next(10)` is executed?
c) Is the `next` method in the `loggingObserver` a pure function?
d) Does the Observer stop executing `next` calls after the `complete` method is called?
e) Comment out the line `clearInterval(timer);`. What happens when you execute the code? Why?

Exercise 6 - SafeObserver

Above you saw an Observable that…. never actually completes. Please fill in the skeleton code for the SafeObserver so that it has similar behaviour to the simple observer, except it completes when `complete` is called. Any further calls to `next` should be ignored. It should also complete when `unsubscribe` is called.

Please complete the SafeObserver class and pass the tests.

Exercise 7 - Your good friends map, filter and forEach

A skeleton Observable has been provided with a constructor, fromArray and subscribe methods. These three provided methods allow us to use this Observable to listen to a simple data stream.

```
Observable.fromArray([1,2,3,4,5,6])
      .subscribe(e => console.log(e));
```

Note that above we don't write `new Observable`. This is because we're calling a static method on the class and don't need to create a new instance of the class.

Please implement map, filter and forEach on your Observable.
Below are the type signatures:

```
map<R>(f: (_:T)=>R): Observable<R>
filter(f: (_:T)=>boolean): Observable<T>
forEach(f: (_:T)=>void): Observable<T>
```

Hint: Look at the `scan` method.

Exercise 8 - Observable.interval method

Instead of observing an array, it would be more interesting to create a static method that returns an Observable that emits the elapsed time since subscribing at increments specified by the user.

Implement a method `interval` on the Observable with the following type signature:

```
interval(milliseconds: number): Observable<number>
```

This will be quite similar to the very simple observable shown in exercise 5.

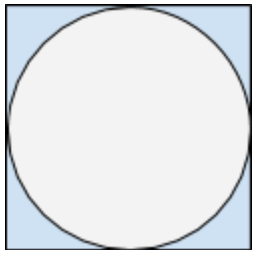
Optional Challenge - Improving a Visualisation of a Monte Carlo approximation of Pi

You're the newest hire on a data visualisation team that's working on visualising the process of approximating Pi using the Monte Carlo method. Congratulations!

Unfortunately some things haven't been finished and you've volunteered to improve the demo! But first, what the heck is a Monte Carlo method!?

The Monte Carlo method is used when a phenomena is easier to simulate than measure. Below we are literally going to randomly place dots on a unit circle and tally up which dots hit the circle and which miss. This will approximate π (because we know the ratio between the circle and square).

Mathematically the area for a circle is $\pi \times radius^2$. Thus the area for a unit circle, or circle with a radius of 1, is π. Let's draw a square around the circle.



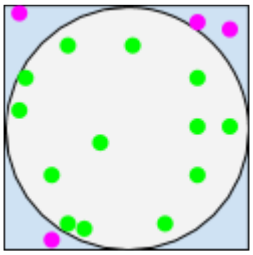
The side of the square is $2 \times radius$ and therefore the area of the square is equal to $(2 \times radius) \times (2 \times radius) = 4 \times radius^2$.

Therefore the ratio between the circle area and square area =
$circle\ area : square\ area$
$\pi \times radius^2 : 4 \times radius^2$
$\pi : 4$

Thus the probability of a random point falling within the circle is $\frac{\pi}{4}$.
Using this probability we can calculate pi!

$$\frac{points\ that\ fell\ within\ circle}{total\ data\ point} = \frac{\pi}{4}$$
$$\frac{points\ that\ fell\ within\ circle}{total\ data\ point} \times 4 = \pi$$

Below is a concrete example with 17 points.



13 green points inside circle.
4 purple points outside circle.
17 total points.

$$\frac{\textit{points that fell within circle}}{\textit{total data point}} \times 4 = \pi$$

$$\frac{13}{17} \times 4 = \pi$$

$$3.05882 = \pi$$

Notice when you open `challenge.html` in the chrome browser that a pi approximation is drawn to the screen, but lacking dots. Please complete the visualisation by drawing dots on the canvas with the dots coloured according to where they land on the canvas. Points within the circle should be a different colour to those outside the circle.

You even found a gif of what the visualisation should look like (*spoilers*):
http://www.giphy.com/gifs/26n6LHbwmraTyBmmY