Principles of Software Construction: Objects, Design, and Concurrency

Immutability, Promises, Patterns

Claire Le Goues Bogdan Vasilescu

Today

- Revisiting Immutability
- Design for Concurrency
  - How to: design for extension, reuse, readability, robustness?
  - The promise (future) pattern
  - Connections to streams, React

Revisiting Immutability

Reading Quiz: Immutability

On Canvas!
Go find it!

Design & Concurrency

- So far, we've introduced a number of low-level idioms/primitives for parallelism and concurrency.
- What are the tradeoffs between them?
- What (design) challenges do we face?

Let’s revisit: Why Immutability?

Let's revisit: Ensuring Immutability

- Don't provide any mutators
- Ensure that no methods may be overridden
- Make all fields final
- Make all fields private
- Ensure security of any mutable components

Immutable?

```
class Stack {
   readonly #inner: any[]
   constructor (inner: any[]) {
       this.#inner=inner
   }
   push(o: any): Stack {
       const newInner = this.#inner.slice()
       newInner.push(o)
       return new Stack(newInner)
   }
   peek(): any {
       return this.#inner[this.#inner.length-1]
   }
   getInner(): any[] {
       return this.#inner
   }
}
```

Immutable?

Inner mutable state (List in Java)

Create copy of mutable object (new ArrayList(old) in Java)

Return new immutable object

```
class Stack {
    readonly #inner: any[]
    constructor (inner: any[]) {
        this.#inner=inner
    }
    push(o: any): Stack {
        const newInner = this.#inner.slice()
        newInner.push(o)
        return new Stack(newInner)
    }
    peek(): any {
        return this.#inner[this.#inner.length-1]
    }
    getInner(): any[] {
        return this.#inner
    }
}
```

Aliasing is what makes Mutable State risky

Many variables may point to same object

Any reference to the object can modify the object, effect seen by all other users

```
const x = [ 1, 2, 3 ]
const y = x
function foo(z: number[]): void { /*...*/ }
foo(y)
```

x, y, and z all point to the same mutable array

Immutable?

Inner mutable state (List in Java)

Create copy of mutable object (new ArrayList(old) in Java)

Return new immutable object

Leak mutable state

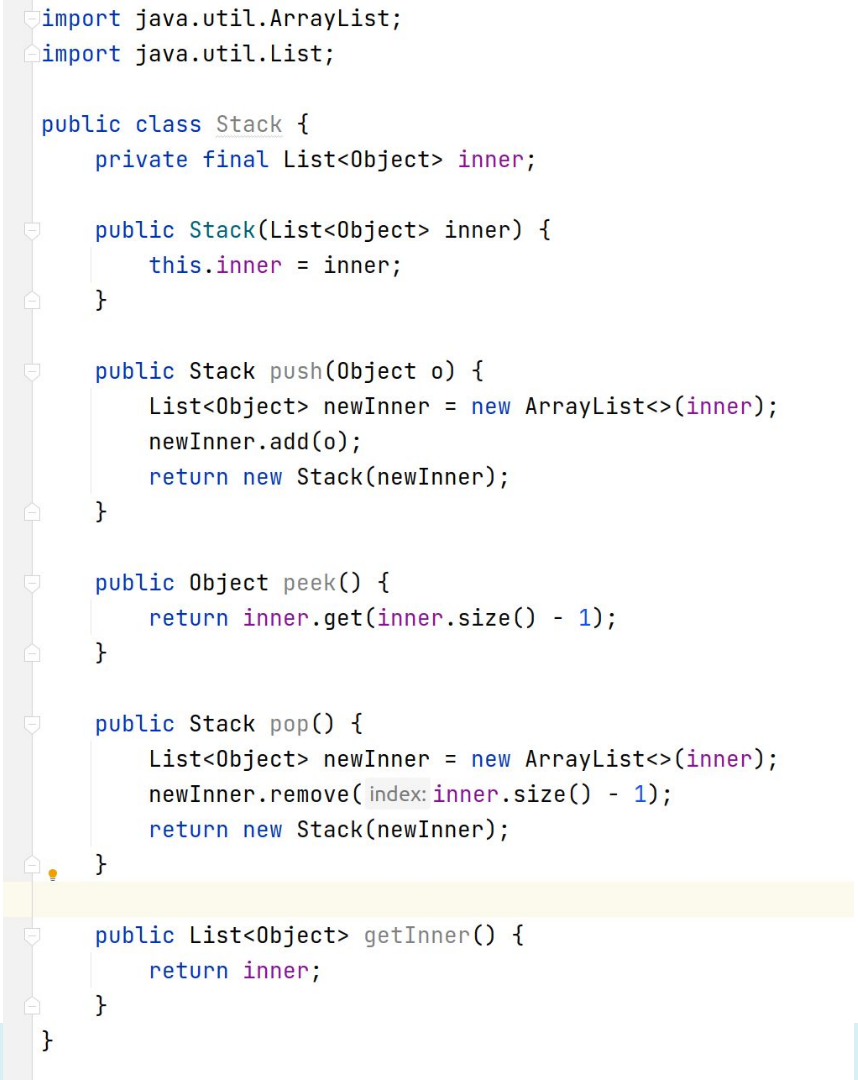
Accept mutable state

```
class Stack {
    readonly #inner: any[]
    constructor (inner: any[]) {
        this.#inner=inner
    }
    push(o: any): Stack {
        const newInner = this.#inner.slice()
        newInner.push(o)
        return new Stack(newInner)
    }
    peek(): any {
        return this.#inner[this.#inner.length-1]
    }
    getInner(): any[] {
        return this.#inner
    }
}
```

Fixed

```
class Stack {
   readonly #inner: any[]
   constructor (inner: any[]) {
       this.#inner=inner.slice()
   }
   push(o: any): Stack {
       const newInner = this.#inner.slice()
       newInner.push(o)
       return new Stack(newInner)
   }
   peek(): any {
       return this.#inner[this.#inner.length-1]
   }
   getInner(): any[] {
       return this.#inner.slice()
       // Java: return new ArrayList(inner)
   }
}
```

(The original in Java)



```java
import java.util.ArrayList;
import java.util.List;

public class Stack {
    private final List<Object> inner;

    public Stack(List<Object> inner) {
        this.inner = inner;
    }

    public Stack push(Object o) {
        List<Object> newInner = new ArrayList<>(inner);
        newInner.add(o);
        return new Stack(newInner);
    }

    public Object peek() {
        return inner.get(inner.size() - 1);
    }

    public Stack pop() {
        List<Object> newInner = new ArrayList<>(inner);
        newInner.remove( index: inner.size() - 1);
        return new Stack(newInner);
    }

    public List<Object> getInner() {
        return inner;
    }
}
```

Recall: Ensuring Immutability

- Don’t provide any mutators
- Ensure that no methods may be overridden
- Make all fields final
- Make all fields private
- **Ensure security of any mutable components**

Writing Immutable Data Structures

Any “set” operation returns a new copy of an object
  (can point to old object to save memory, e.g. linked lists)

Final fields of immutable objects are save (e.g., strings, numbers)

Fields of mutable objects must be protected
  (encapsulation, making copies)

Careful with mutable constructor/method arguments (make copies)

Easy to make mistakes when mixing mutable and immutable data structures, only academic tools for checking

Trend toward immutable data structures

Immutable data structures common in functional programming

Many recent languages and libraries embrace immutability

- Scala, Rust, stream, React, Java Records

Simplifies building concurrent and distributed systems

Requires some practice when used to imperative programming with mutable state, but will become natural

Circular references & Caching

Immutable data structures often from a directed acyclic graph

Cycles challenging

Cycles often useful for performance (caching)

```
class TreeNode {
    readonly #parent: TreeNode
    readonly #children: TreeNode[]
    constructor(parent: TreeNode,
                children: TreeNode[]) {
        this.#parent = parent
        this.#children = children
    }
    addChild(child: TreeNode) {
        const newChildren = this.#children.slice()
        //const newChild = child.setParent(this)  ??
        newChildren.push(child)
        const newNode = new TreeNode(this.#parent,
                                     newChildren)
        //child.setParent(newNode)  ??
        return newNode
    }
}
```

Design Discussion

Design for Understandability / Maintainability

- Immutable objects are easy to reason about, they won’t change
- Mutable objects have more complicated contracts, function and client both can modify state
- Do not need to think about corner cases of concurrent modification

Design for Reuse

- Easy to reuse even in concurrent settings

Java 16 Records

Records are (shallowly) immutable

No setters

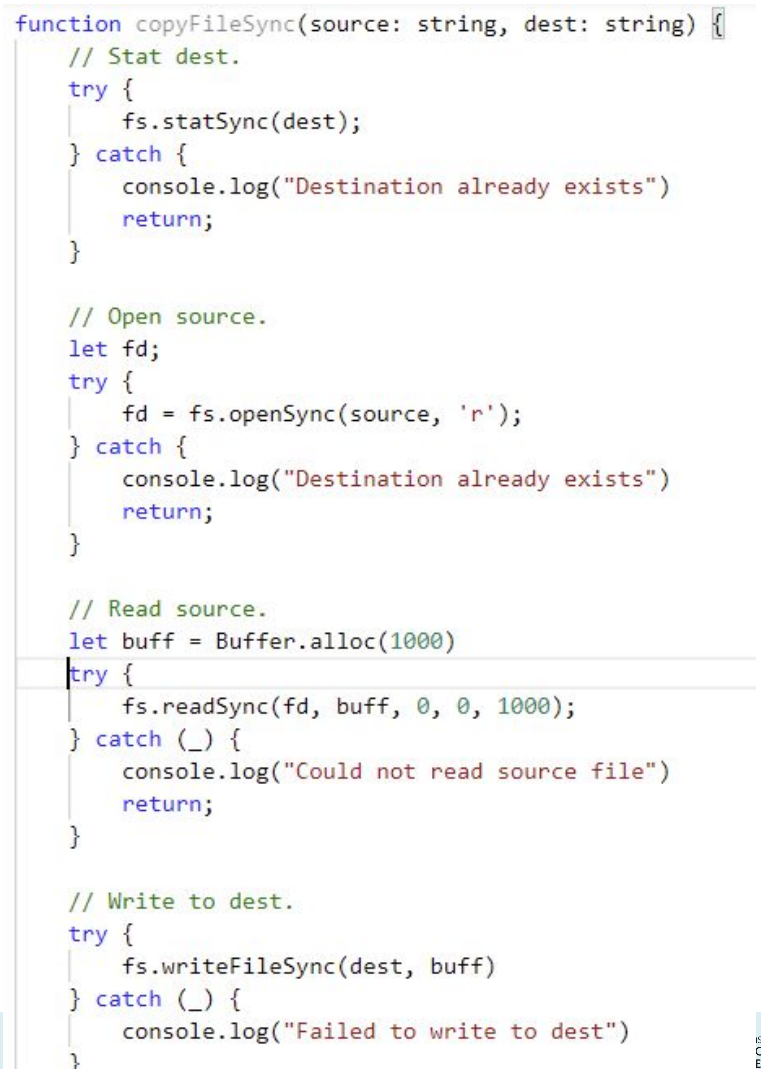
But also no defensive copying of mutable fields

Design Goals: Concurrency

- What are we looking for in design?
  - Reuse
  - **Readability**
  - **Robustness**
  - Extensibility
  - Performance
  - ...

A simple function

...in sync world



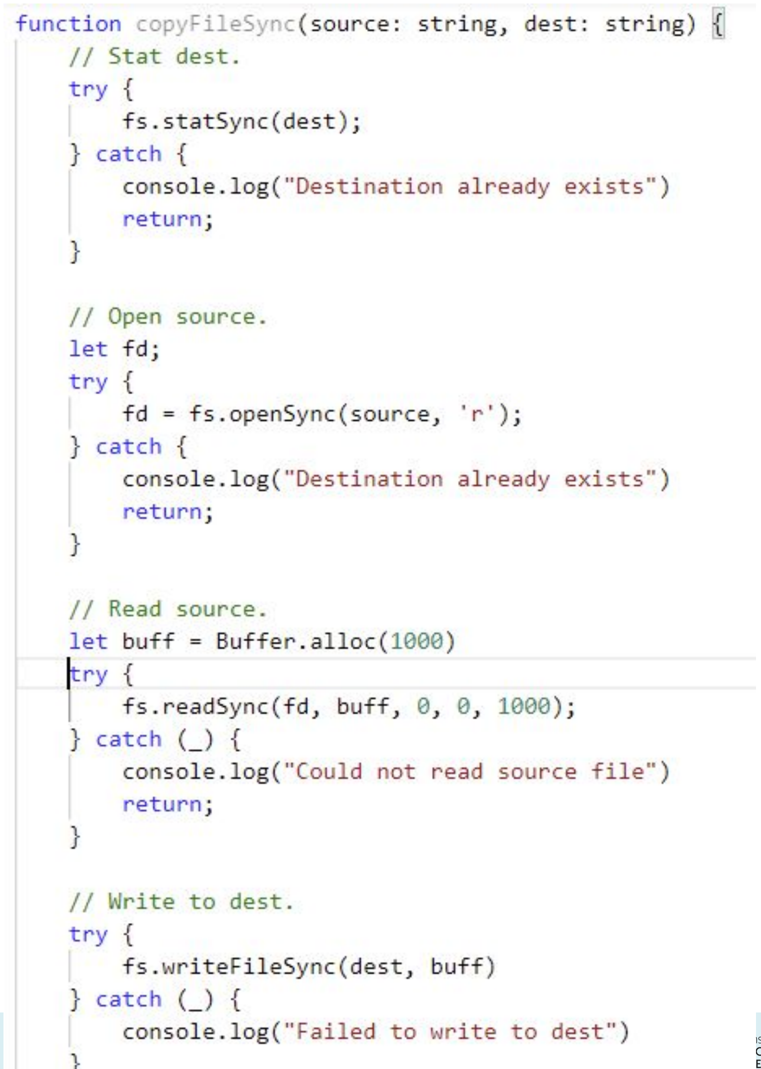
```
function copyFileSync(source: string, dest: string) {
    // Stat dest.
    try {
        fs.statSync(dest);
    } catch {
        console.log("Destination already exists")
        return;
    }

    // Open source.
    let fd;
    try {
        fd = fs.openSync(source, 'r');
    } catch {
        console.log("Destination already exists")
        return;
    }

    // Read source.
    let buff = Buffer.alloc(1000)
    try {
        fs.readSync(fd, buff, 0, 0, 1000);
    } catch (_) {
        console.log("Could not read source file")
        return;
    }

    // Write to dest.
    try {
        fs.writeFileSync(dest, buff)
    } catch (_) {
        console.log("Failed to write to dest")
    }
```

(Code example)

A simple function...in sync world

How to make this asynchronous?

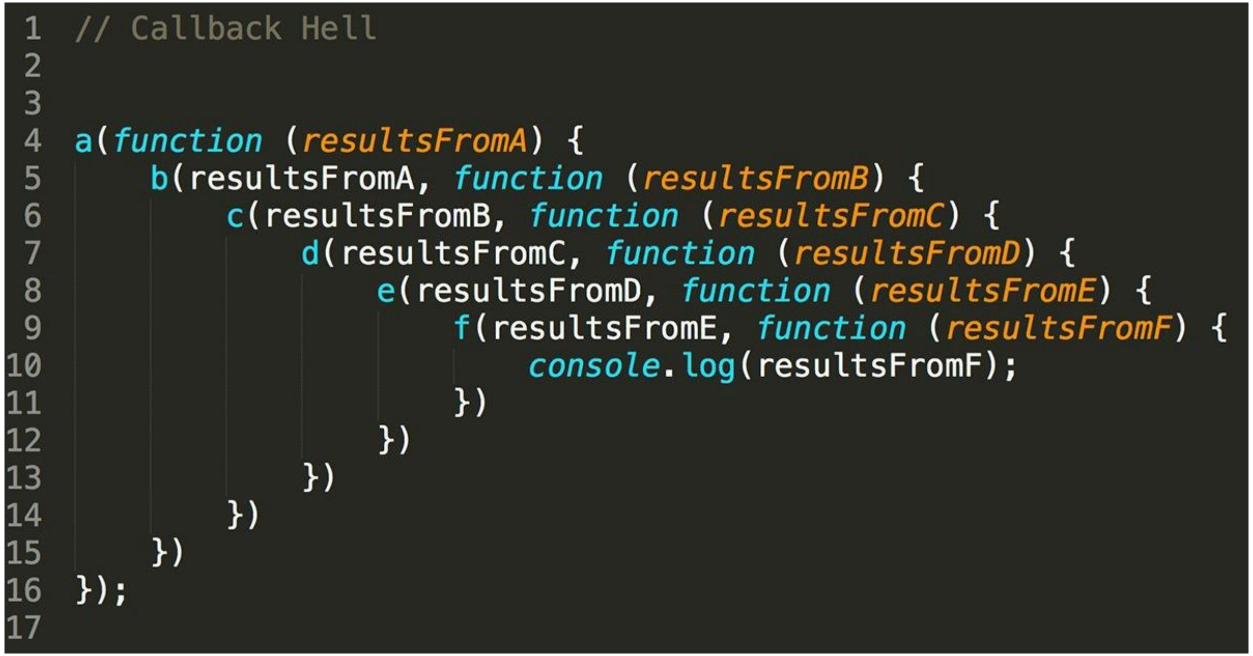
- What needs to "happen first"?
- What is the control-flow in callback world?



```
function copyFileSync(source: string, dest: string) {
    // Stat dest.
    try {
        fs.statSync(dest);
    } catch {
        console.log("Destination already exists")
        return;
    }

    // Open source.
    let fd;
    try {
        fd = fs.openSync(source, 'r');
    } catch {
        console.log("Destination already exists")
        return;
    }

    // Read source.
    let buff = Buffer.alloc(1000)
    try {
        fs.readSync(fd, buff, 0, 0, 1000);
    } catch (_) {
        console.log("Could not read source file")
        return;
    }

    // Write to dest.
    try {
        fs.writeFileSync(dest, buff)
    } catch (_) {
        console.log("Failed to write to dest")
    }
```

Event Handling in JS: Callback Hell

What if our callbacks need callbacks?



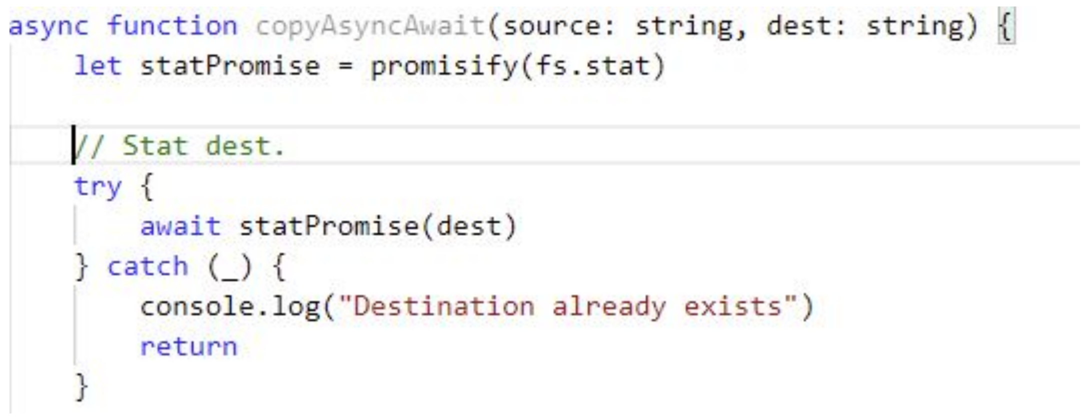
```
// Callback Hell


a(function (resultsFromA) {
    b(resultsFromA, function (resultsFromB) {
        c(resultsFromB, function (resultsFromC) {
            d(resultsFromC, function (resultsFromD) {
                e(resultsFromD, function (resultsFromE) {
                    f(resultsFromE, function (resultsFromF) {
                        console.log(resultsFromF);
                    })
                })
            })
        })
    })
});

```

Promises

- Are immutable
- And available repeatedly to observers
- Compare ‘Future’ in Java
  - ‘CompletableFuture’ is probably closest
- Downsides:
  - Still heavy syntax
  - Hard to trace errors
  - Doesn’t quite solve complex callbacks
    - E.g., if X, call this, else that

Next Step: Async/Await

- Async functions return a promise
  - May wrap concrete values
  - May return rejected promises on exceptions
- Allowed to ‘await’ synchronously



```
async function copyAsyncAwait(source: string, dest: string) {
    let statPromise = promisify(fs.stat)

    // Stat dest.
    try {
        await statPromise(dest)
    } catch (_) {
        console.log("Destination already exists")
        return
    }
```

Design Goals

- What are we looking for in design?
    - *Reuse*
    - **Readability**
    - **Robustness**
    - **Extensibility**
    - **Performance**
    - ...

The Promise Pattern

- Problem: one or more values we will need will arrive later
  - At some point we <u>must</u> wait
- Solution: an abstraction for *expected values*
- Consequences:
  - Declarative behavior for when results become available (*conf.* callbacks)
  - Need to provide paths for normal and abnormal execution
    - E.g., then() and catch()
  - May want to allow combinators
  - Debugging requires some rethinking

Promises: Guarantees

- Callbacks are never invoked before the current run of the event loop completes
- Callbacks are always invoked, even if (chronologically) added after asynchronous operation completes
- Multiple callbacks are called in order

Design for Concurrency

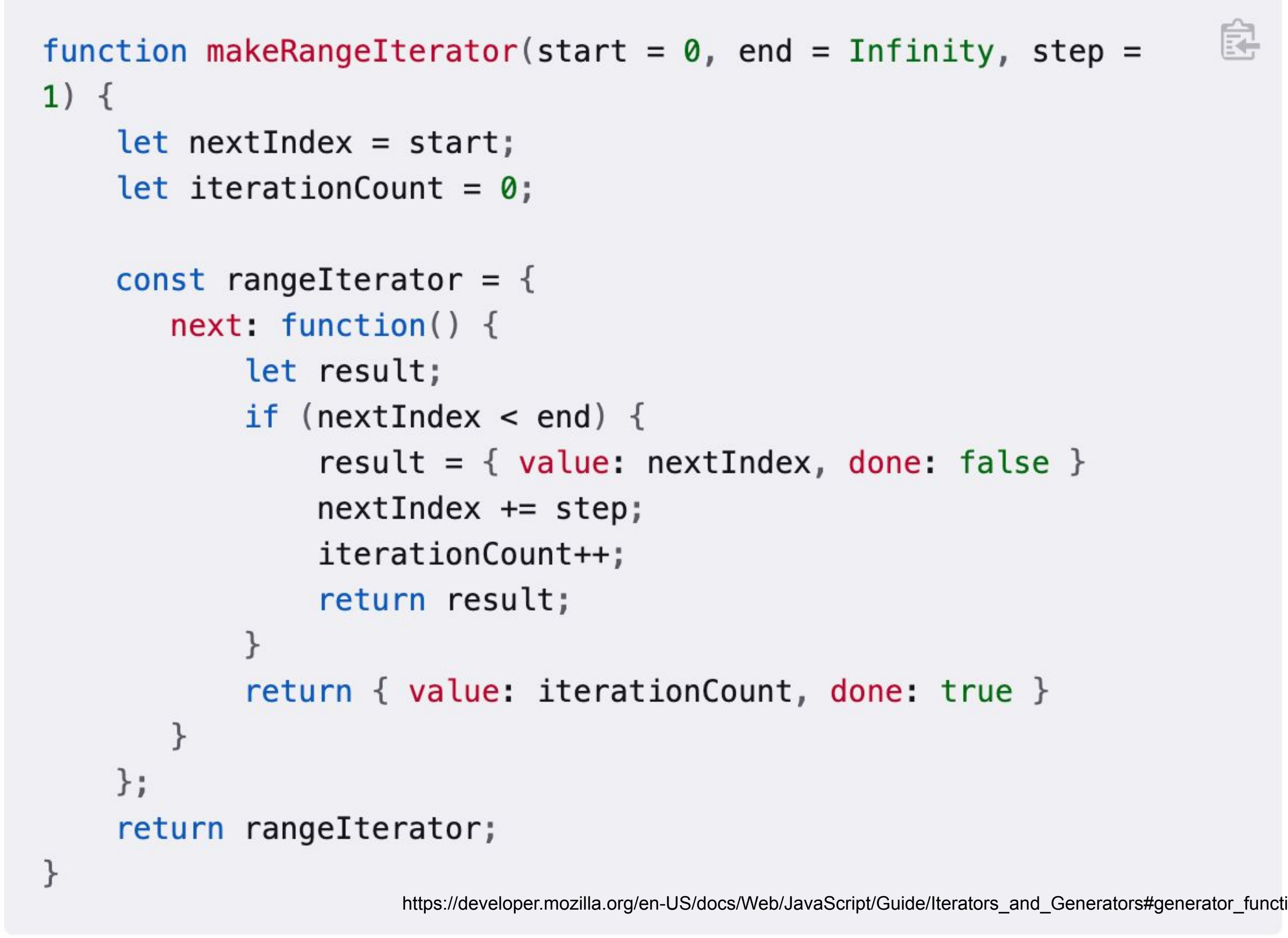
Let’s squint at a few similar developments

Generator Pattern

- Problem: process a collection of indeterminate size
- Solution: provide data points on request when available
- Consequences:
  - Each call to ‘next’ is like awaiting a promise
  - A generator can be infinite, and can announce if it is complete.
  - Generators can be *lazy*, only producing values on demand
    - Or producing promises

(quick code example)

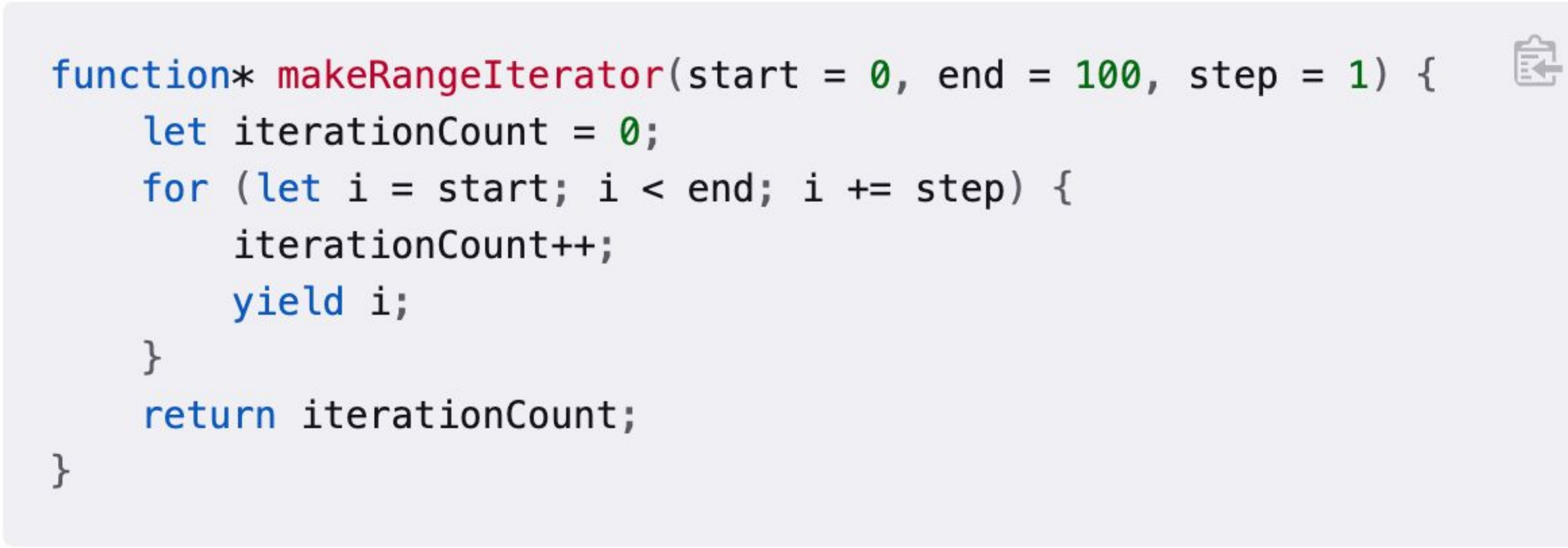
Note that Generators also exist in Java!



```
function makeRangeIterator(start = 0, end = Infinity, step =
1) {
    let nextIndex = start;
    let iterationCount = 0;

    const rangeIterator = {
       next: function() {
           let result;
           if (nextIndex < end) {
               result = { value: nextIndex, done: false }
               nextIndex += step;
               iterationCount++;
               return result;
           }
           return { value: iterationCount, done: true }
       }
    };
    return rangeIterator;
}
```

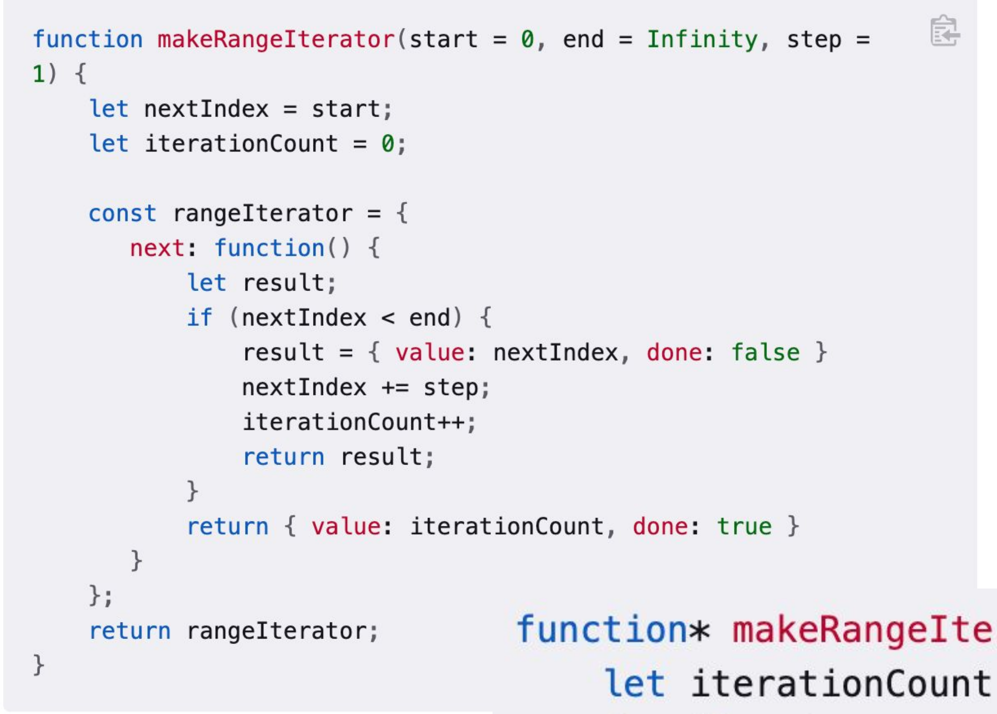
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Iterators_and_Generators#generator_functions

```
const it = makeRangeIterator(1, 10, 2);

let result = it.next();
while (!result.done) {
 console.log(result.value); // 1 3 5 7 9
 result = it.next();
}

console.log("Iterated over sequence of size: ", result.value);
// [5 numbers returned, that took interval in between: 0 to
10]
```

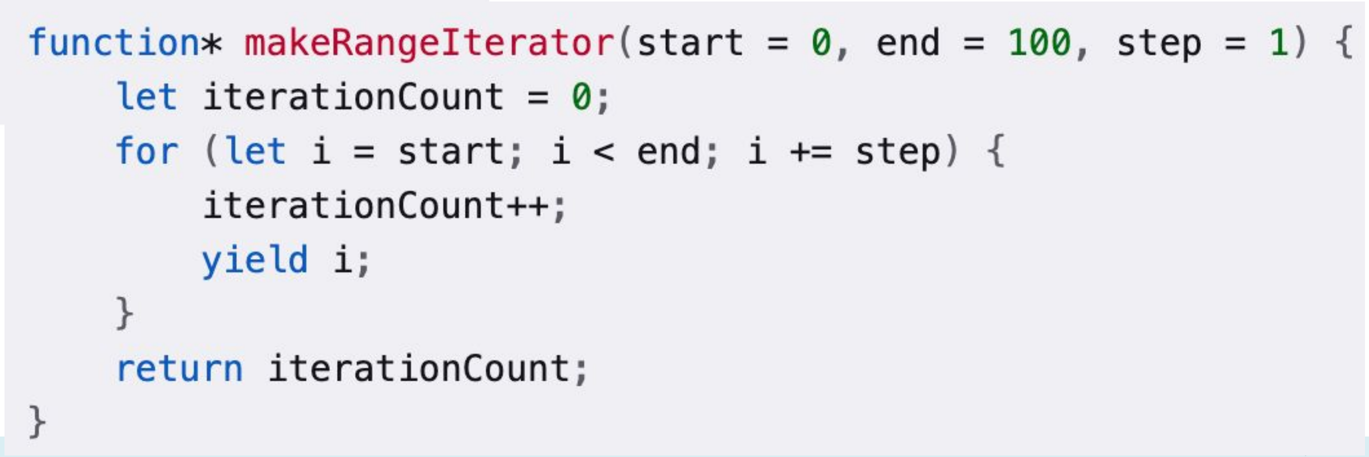


```
function* makeRangeIterator(start = 0, end = 100, step = 1) {
    let iterationCount = 0;
    for (let i = start; i < end; i += step) {
        iterationCount++;
        yield i;
    }
    return iterationCount;
}
```

Tradeoffs?



```
function makeRangeIterator(start = 0, end = Infinity, step =
1) {
    let nextIndex = start;
    let iterationCount = 0;

    const rangeIterator = {
       next: function() {
           let result;
           if (nextIndex < end) {
               result = { value: nextIndex, done: false }
               nextIndex += step;
               iterationCount++;
               return result;
           }
           return { value: iterationCount, done: true }
       }
    };
    return rangeIterator;
}
```



```
function* makeRangeIterator(start = 0, end = 100, step = 1) {
    let iterationCount = 0;
    for (let i = start; i < end; i += step) {
        iterationCount++;
        yield i;
    }
    return iterationCount;
}
```

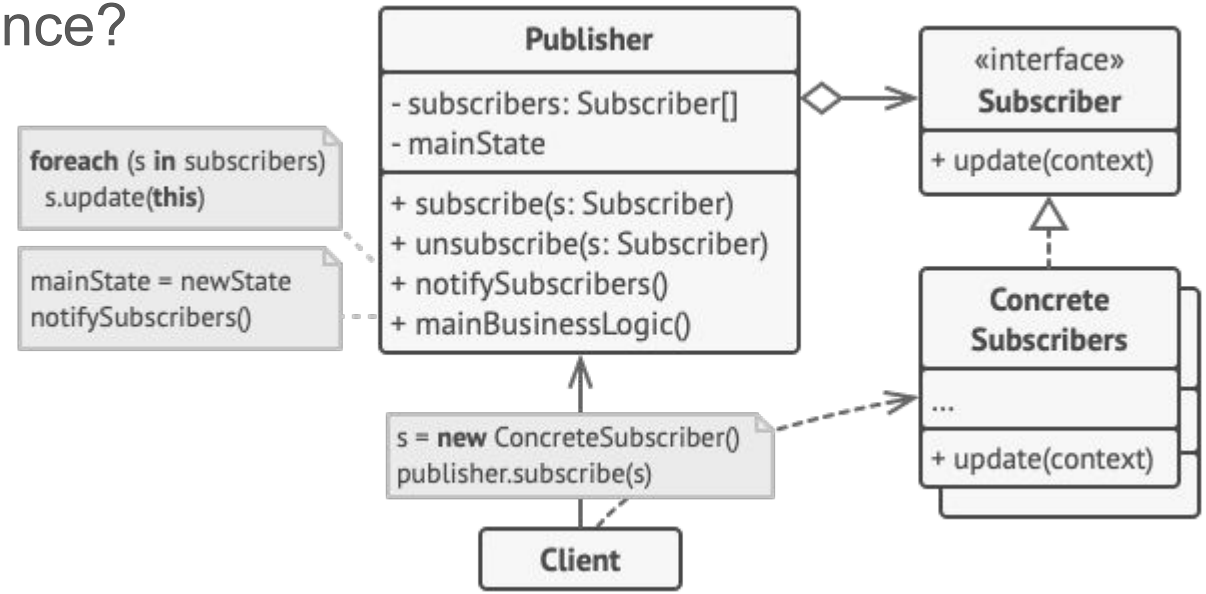
Generator Pattern

- Problem: process a collection of indeterminate size
- Solution: provide data points on request when available
- Consequences:
  - Each call to ‘next’ is like awaiting a promise
  - A generator can be infinite, and can announce if it is complete.
  - Generators can be *lazy*, only producing values on demand
    - Or producing promises
- Where might this be useful?

Observer Pattern

Recall: let objects observe behavior of others

What is the difference?



```
foreach (s in subscribers)
  s.update(this)
```

```
mainState = newState
notifySubscribers()
```

```
s = new ConcreteSubscriber()
publisher.subscribe(s)
```

https://refactoring.guru/design-patterns/observer

Observer vs. Generator

Push vs. Pull

- In Observer, the publisher controls information flow
  - When it pushes, everyone must listen
- In generators, the listener “pulls” elements
  - Generator may only prepare the next element upon/after pull
- Which is better?
  - Generators are in a sense ‘observers’ to their clients.
  - This inversion of control can make flow management easier

Manipulating Data

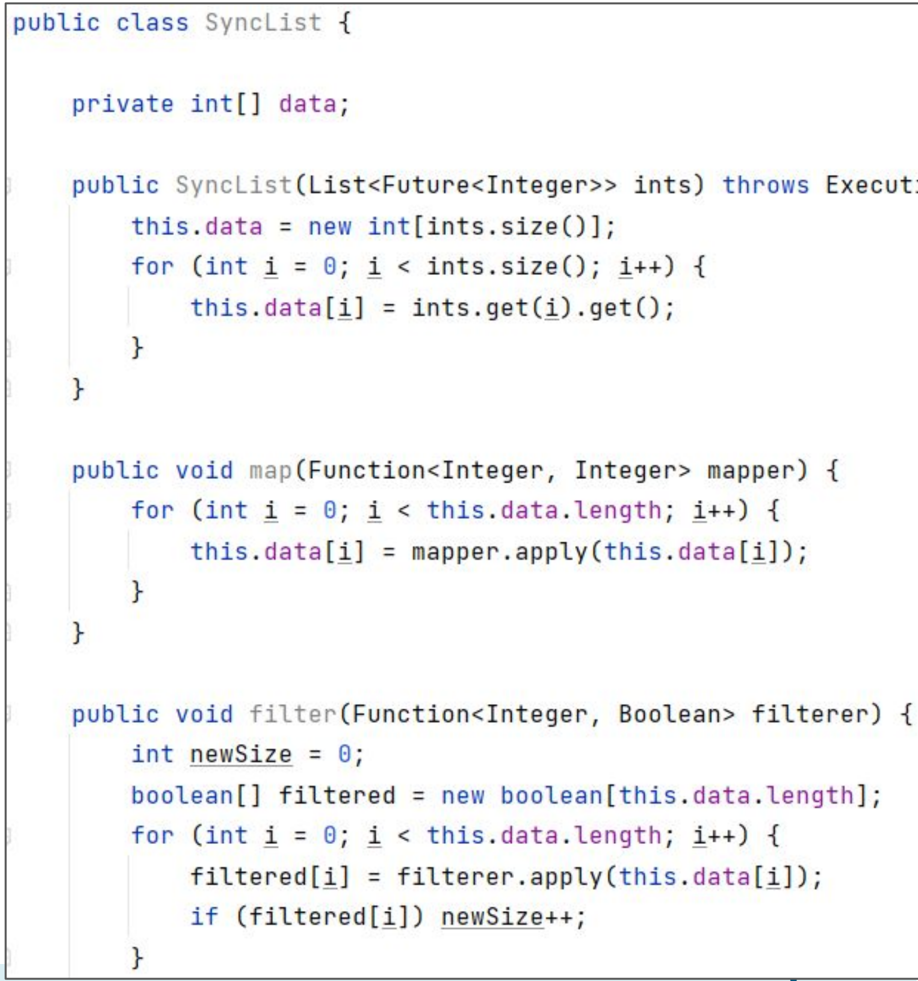
Problem: processing sequential data without assuming its presence

- Let's assume a list of future ints
- Apply a series of transformations
  - E.g., map/update, filter
- Use the result in some operation
  - E.g., collect, foreach

Manipulating Data

Easy solution: collect it all

- Downsides?



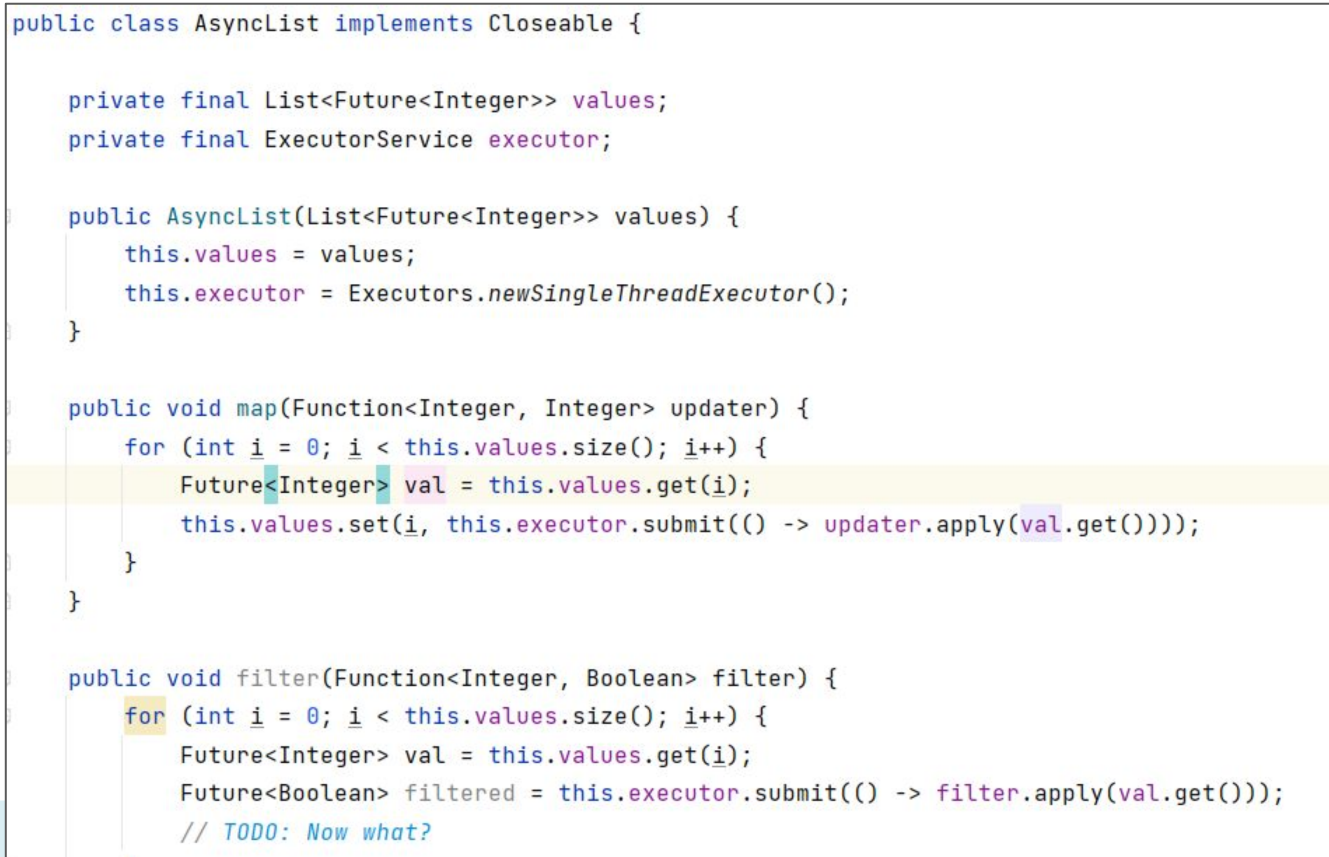
```java
public class SyncList {

    private int[] data;

    public SyncList(List<Future<Integer>> ints) throws Executi
        this.data = new int[ints.size()];
        for (int i = 0; i < ints.size(); i++) {
            this.data[i] = ints.get(i).get();
        }
    }

    public void map(Function<Integer, Integer> mapper) {
        for (int i = 0; i < this.data.length; i++) {
            this.data[i] = mapper.apply(this.data[i]);
        }
    }

    public void filter(Function<Integer, Boolean> filterer) {
        int newSize = 0;
        boolean[] filtered = new boolean[this.data.length];
        for (int i = 0; i < this.data.length; i++) {
            filtered[i] = filterer.apply(this.data[i]);
            if (filtered[i]) newSize++;
        }
```

Design Goals

- What are we looking for in design?
  - Reuse
  - **Readability**
  - Robustness
  - **Extensibility**
  - **Performance**
  - ...

Manipulating Data

How about:



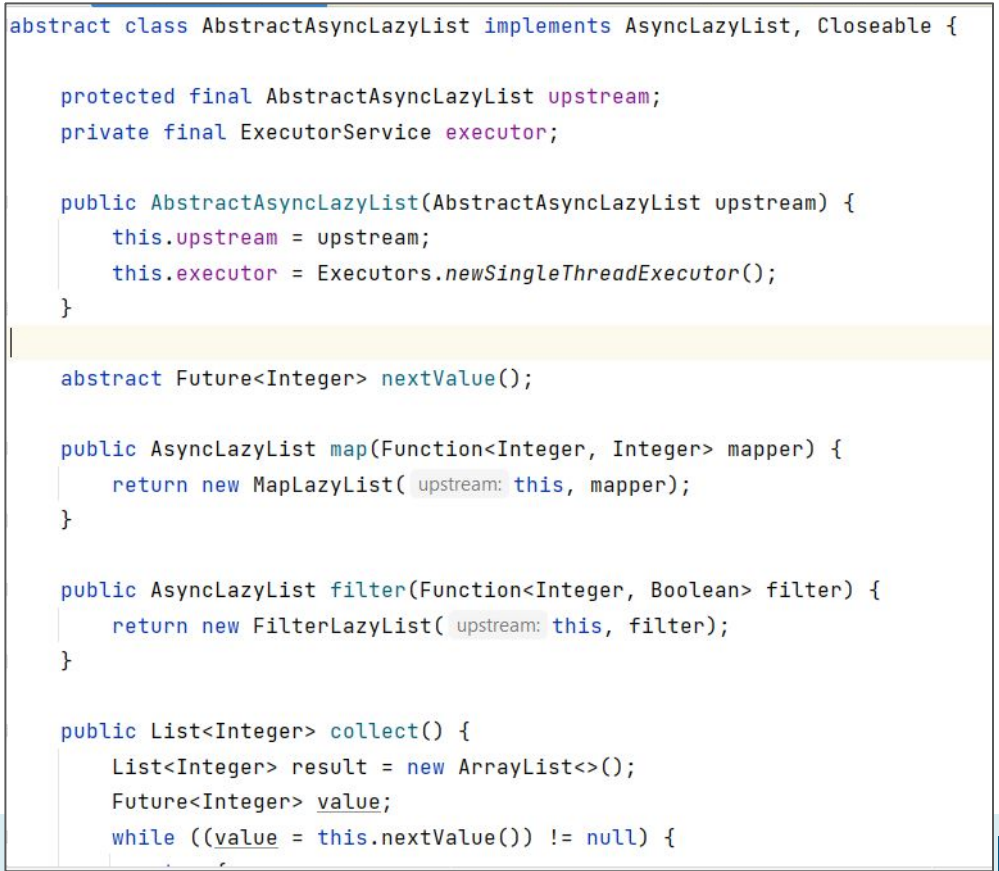
```java
public class AsyncList implements Closeable {

    private final List<Future<Integer>> values;
    private final ExecutorService executor;

    public AsyncList(List<Future<Integer>> values) {
        this.values = values;
        this.executor = Executors.newSingleThreadExecutor();
    }

    public void map(Function<Integer, Integer> updater) {
        for (int i = 0; i < this.values.size(); i++) {
            Future<Integer> val = this.values.get(i);
            this.values.set(i, this.executor.submit(() -> updater.apply(val.get())));
        }
    }

    public void filter(Function<Integer, Boolean> filter) {
        for (int i = 0; i < this.values.size(); i++) {
            Future<Integer> val = this.values.get(i);
            Future<Boolean> filtered = this.executor.submit(() -> filter.apply(val.get()));
            // TODO: Now what?
```

Design Goals

- What are we looking for in design?
  - **Reuse**
  - **Readability**
  - *Robustness*
  - **Extensibility**
  - **Performance**
  - …

Manipulating Data

How about:



```
abstract class AbstractAsyncLazyList implements AsyncLazyList, Closeable {

    protected final AbstractAsyncLazyList upstream;
    private final ExecutorService executor;

    public AbstractAsyncLazyList(AbstractAsyncLazyList upstream) {
        this.upstream = upstream;
        this.executor = Executors.newSingleThreadExecutor();
    }

    abstract Future<Integer> nextValue();

    public AsyncLazyList map(Function<Integer, Integer> mapper) {
        return new MapLazyList( upstream: this, mapper);
    }

    public AsyncLazyList filter(Function<Integer, Boolean> filter) {
        return new FilterLazyList( upstream: this, filter);
    }

    public List<Integer> collect() {
        List<Integer> result = new ArrayList<>();
        Future<Integer> value;
        while ((value = this.nextValue()) != null) {
```

Design Goals

- What are we looking for in design?
  - **Reuse**
  - **Readability**
  - Robustness
  - **Extensibility**
  - **Performance**
  - ...

Remember Iterators and Streams?

Iterate over elements in arbitrary data structures (lists, sets, trees) without having to know internals

Typical interface:

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

(in Java also **remove**)

Iterator design pattern

- Problem: Clients need uniform strategy to access all elements in a container, independent of the container type
    - Order is unspecified, but access every element once
- Solution: A strategy pattern for iteration
- Consequences:
    - Hides internal implementation of underlying container
    - Easy to change container type
    - Facilitates communication between parts of the program

Streams

- Stream: generators for Java!
  - stream.generate() → infinite stream
  - A sequence of objects
  - *Not* interested in accessing specific addresses
- Typically provide operations
  - To translate stream: map, flatMap, filter
  - Operations on all elements (fold, sum) with higher-order functions
  - Often provide efficient/parallel implementations (subtype polymorphism)
- Built-in in Java since Java 8; basics in Node libraries in JS

Summary

- Concurrency brings unique design problems
  - And patterns
  - Promises are a key one
  - Worth understanding relations to (async) generators, streams