Principles of Software Construction: Objects, Design, and Concurrency

Concurrency: Patterns & Promises

Christian Kästner **Vincent Hellendoorn**

Carnegie Mellon University School of Computer Science

institute for SOFTWARE RESEARCH

Today

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

Design Goals

- What are we looking for in design?

Design Goals

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

Design & Concurrency

- So far, mostly **low-level** concurrency idioms
  - What design challenges do we face?
- Two case-studies
  - Code examples off-slides

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")
    }
```

Design Goals

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

Design Goals

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

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

What if our callbacks need callbacks?

Callback Hell

More than nested functions!

- How to handle conditions (or loops)?
- Managing exceptional behavior in both sync and async code

Design Goals

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

Design Goals

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

Promises

- Are immutable
- And available repeatedly to observers
- Compare ‘Future’ in Java
  - ‘CompletableFuture’ is probably closest

Design Goals

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

Design Goals

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

Promises: 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
  - ...

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 must 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

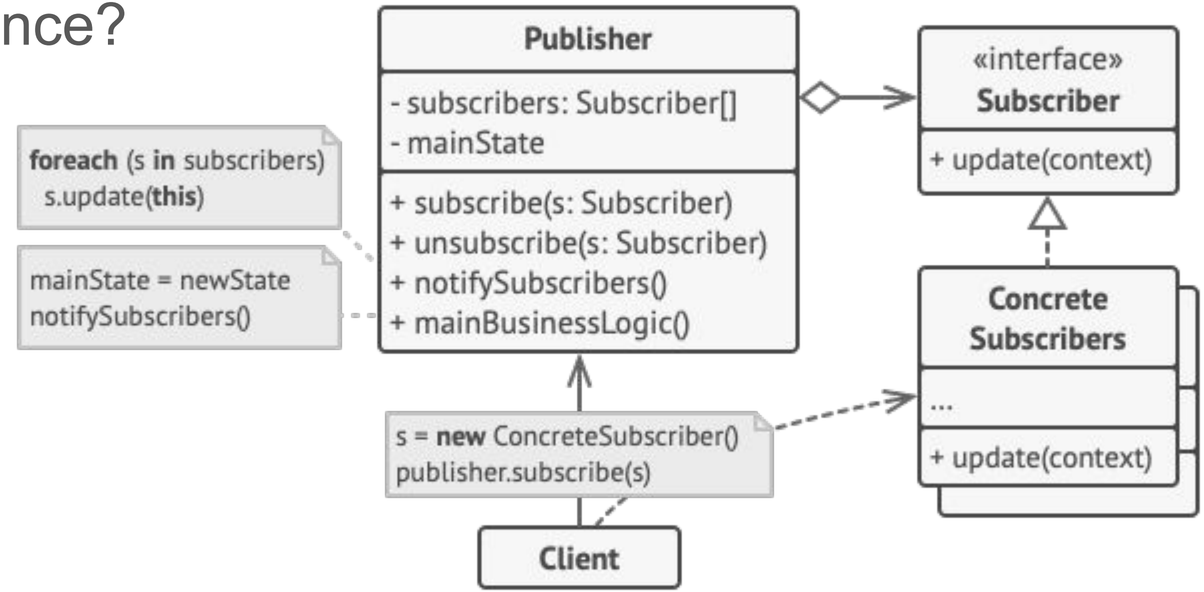
Generator Pattern

- 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
- Where might this be useful?

Observer Pattern

Recall: let objects observe behavior of others

What is the difference?



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 Execut
        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()));
```

Design Goals

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

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
  - ...

Design Goals

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

Traversing a collection

- Since Java 1.0:

```java
  Vector arguments = …;
  for (int i = 0; i < arguments.size(); ++i) {
    System.out.println(arguments.get(i));
  }
```

- Java 1.5: enhanced for loop

```java
List<String> arguments = …;
for (String s : arguments) {
  System.out.println(s);
}
```

- Works for every implementation of `Iterable`

```java
public interface Iterable<E> {
  public Iterator<E> iterator();
}
public interface Iterator<E> {
  boolean hasNext();
  E next();
  void remove();
```

- In JavaScript (ES6)

```javascript
let arguments = …
for (const s of arguments) {
  console.log(s)
}
```

- Works for every implementation with a “magic” function `[Symbol.iterator]` providing an iterator

```typescript
interface Iterator<T> {
  next(value?: any): IteratorResult<T>;
  return?(value?: any): IteratorResult<T>;
  throw?(e?: any): IteratorResult<T>;
}
interface IteratorReturnResult<TReturn> {
  done: true;
  value: TReturn;
}
```

The Iterator Idea

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)

Using an iterator

Can be used explicitly

```
List<String> arguments = …;
for (Iterator<String> it = arguments.iterator(); it.hasNext();  ) {
  String s = it.next();
  System.out.println(s);
}
```

Often used with magic syntax:

```
for (String s : arguments)
for (const s of arguments)
```

Java: Getting an Iterator

```
public interface Collection<E> extends Iterable<E> {
  boolean     add(E e);
  boolean     addAll(Collection<? extends E> c);
  boolean     remove(Object e);
  boolean     removeAll(Collection<?> c);
  boolean     retainAll(Collection<?> c);
  boolean     contains(Object e);
  boolean     containsAll(Collection<?> c);
  void        clear();
  int         size();
  boolean     isEmpty();
  Iterator<E> iterator();
  Object[]    toArray()
  <T> T[]     toArray(T[] a);
  …
}
```

Defines an interface for creating an Iterator,
but allows Collection implementation to decide which Iterator to create.

Iterators for everything

```java
public class Pair<E> {
  private final E first, second;
  public Pair(E f, E s) { first = f; second = s; }







}
```

```java
Pair<String> pair = new Pair<String>("foo", "bar");
for (String s : pair) { … }
```

An Iterator implementation for Pairs

```
public class Pair<E> implements Iterable<E> {
  private final E first, second;
  public Pair(E f, E s) { first = f; second = s; }
  public Iterator<E> iterator() {
    return new PairIterator();
  }
  private class PairIterator implements Iterator<E> {
    private boolean seenFirst = false, seenSecond = false;
    public  boolean hasNext() { return !seenSecond; }
    public  E next() {
      if (!seenFirst)  { seenFirst  = true; return first;  }
      if (!seenSecond) { seenSecond = true; return second; }
      throw new NoSuchElementException();
    }
    public void remove() {
      throw new UnsupportedOperationException();
    }
  }
}
```

```
Pair<String> pair = new Pair<String>("foo", "bar");
for (String s : pair) { … }
```

ıtute for
FTWARE
SEARCH

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 is like an Iterator
  - 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

```java
List<String>results = stream.map(Object::toString)
                .filter(s -> pattern.matcher(s).matches())
                .collect(Collectors.toList());
```

```java
int sum = numbers.parallelStream().reduce(0, Integer::sum);
```

```javascript
for (let [odd, even] in numbers.split(n => n % 2, n => !(n % 2)).zip()) {
    console.log(`odd = ${odd}, even = ${even}`);  // [1, 2], [3, 4], ...
 }
```

```javascript
Stream(people).filter({age: 23}).flatMap("children").map("firstName")
   .distinct().filter(/a.*/i).join(", ");
```

A Glimpse at Reactive Programming

(not to be confused with ReactJS)

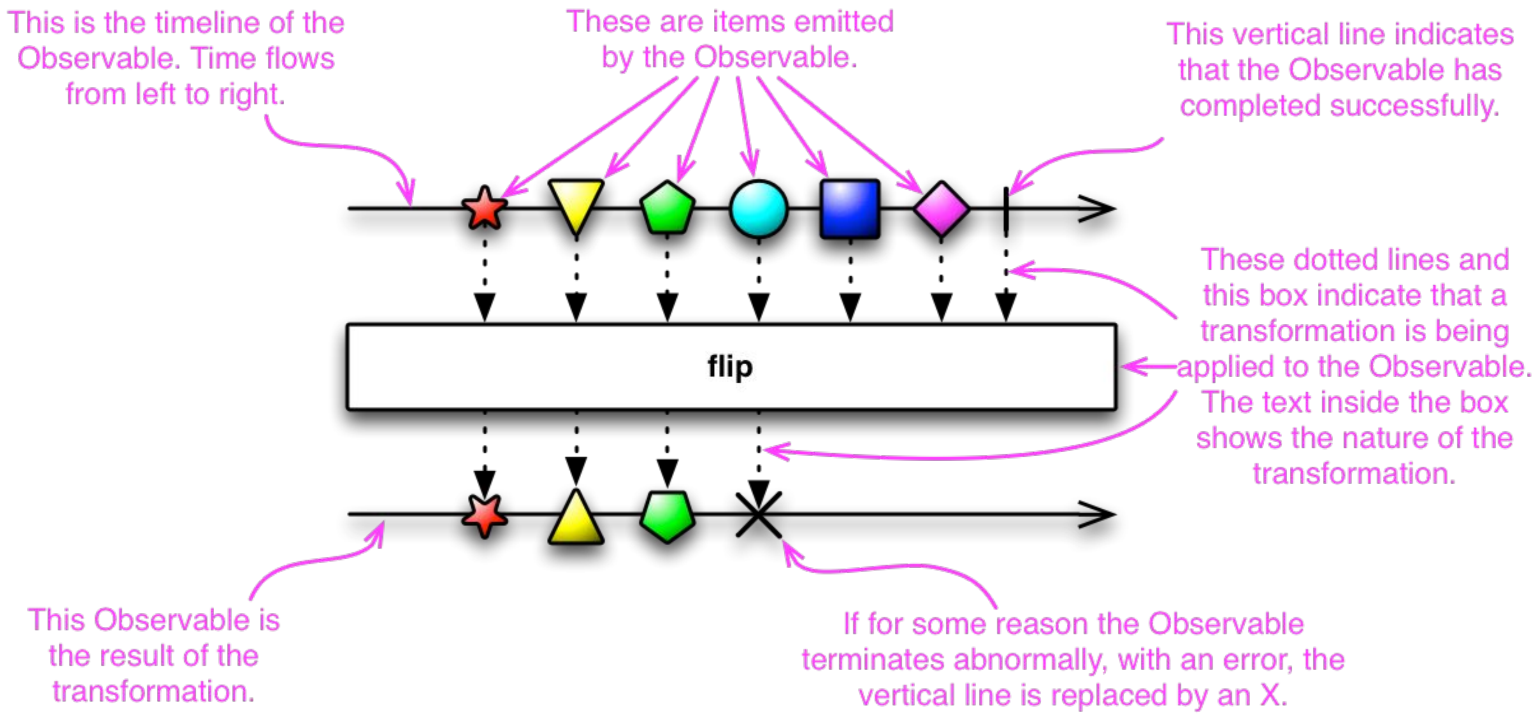
Observable

Observables are lazy Push collections of multiple values. They fill the missing spot in the following table:

	SINGLE	MULTIPLE
Pull	`Function`	`Iterator`
Push	`Promise`	`Observable`

https://rxjs.dev/guide/observable

A Glimpse at Reactive Programming



http://reactivex.io/documentation/observable.html

A Glimpse at Reactive Programming

(not to be confused with ReactJS)

- Rx Observables
  - Similar to “standard” observers
    - “An Observable is just the Observer pattern with a jetpack”*
  - Combined with a rich set of *operators*
    - Compare the stream library, times a lot
  - And *flow-control* in the form of back-pressure
  - Makes for a unified API for polymorphic asynchronous events

*https://x-team.com/blog/rxjs-observables/

Summary

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

Self-Assess In-Class Participation

https://bit.ly/214selfpart

Designing for Concurrency

- Previously: synchronization of methods, variable read/writes
  - Is that enough?

Designing for Concurrency

- Previously: synchronization of methods, variable read/writes
    - Is that enough?

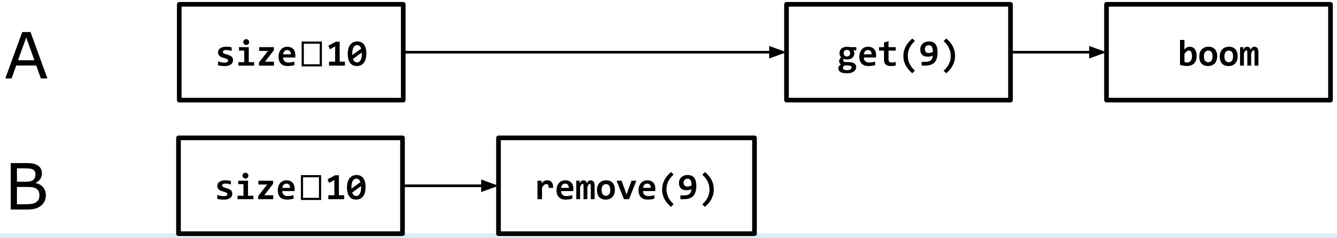
```
public static Object getLast(Vector list) {
    int lastIndex = list.size() - 1;
    return list.get(lastIndex);
}

public static void deleteLast(Vector list) {
    int lastIndex = list.size() - 1;
    list.remove(lastIndex);
}
```

Object-level concurrency

```
public static Object getLast(Vector list) {
    int lastIndex = list.size() - 1;
    return list.get(lastIndex);
}

public static void deleteLast(Vector list) {
    int lastIndex = list.size() - 1;
    list.remove(lastIndex);
}
```



Object-level concurrency

Client-side synchronization

```
public static Object getLast(Vector list) {
    synchronized (list) {
        int lastIndex = list.size() - 1;
        return list.get(lastIndex);
    }
}

public static void deleteLast(Vector list) {
    synchronized (list) {
        int lastIndex = list.size() - 1;
        list.remove(lastIndex);
    }
}
```

Object-level concurrency

What is the risk here?

```
for (int i = 0; i < vector.size(); i++)
    doSomething(vector.get(i));
```

Object-level concurrency

What is the risk here?

```
for (int i = 0; i < vector.size(); i++)
    doSomething(vector.get(i));


synchronized (vector) {
    for (int i = 0; i < vector.size(); i++)
        doSomething(vector.get(i));
}
```

Object-level concurrency

A common mistake:

```
public Object setup() {
  if (obj == null) {
    synchronized (this) {
      obj = this.initializeObject()
    }
  }
}
```