Reactive Programming

1. The need for reactive programming
2. The promise of reactive programming
3. The usage of reactive programming

Demo project: **`demo-reactive`**

[Foreword] Project Lombok (1 of 2)

- The examples in this chapter make use of Project Lombok
  - Via the following pom dependency

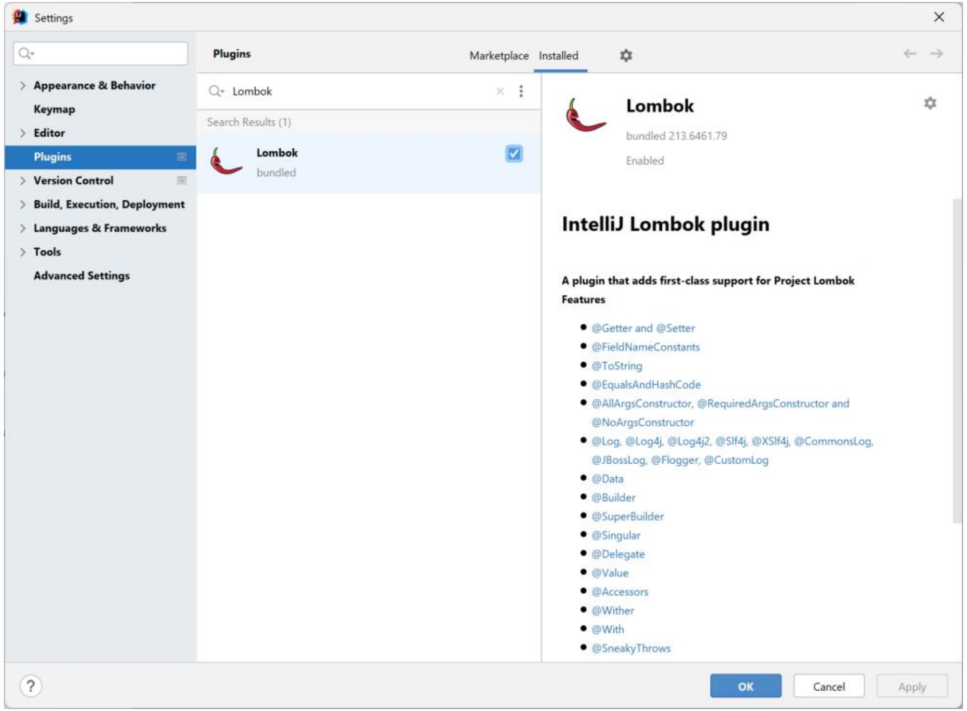
```
<dependencies>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>
    …
</dependencies>
```

pom.xml in demo project

- Project Lombok defines annotations to simplify your code
  - It can generate getters/setters, constructors, `toString()` etc.
  - For full details, see https://projectlombok.org/features/all

[Foreword] Project Lombok (2 of 2)

- You must also install the Project Lombok plugin in IntelliJ
  - Via File | Settings | Plugins



1. The Need for Reactive Programming

- What is reactive programming
- Doing synchronous I/O
- Characteristics of synchronous I/O
- Asynchronous I/O to the rescue
- Aside: Java NIO
- Doing asynchronous I/O
- Characteristics of asynchronous I/O

What is Reactive Programming?

- Reactive programming is a way to process asynchronous data streams
  - Asynchronous I/O can offer big improvements in performance
  - Avoid wasting CPU cycles that are idly waiting for I/O to complete
- Reactive programming inverts the way we do I/O
  - Rather than the client asking for data from the server, the client is notified when data arrives
  - This enables the client to do other work in the meantime
- Reactive programming is a pub/sub pattern
  - Publisher publishes a stream of data
  - Subscriber subscribes to stream and receives data asynchronously

Doing Synchronous I/O (1 of 4)

- To appreciate the need for asynchronous I/O, it's useful to first see the problems with synchronous I/O
  - We'll see how to read a file synchronously
  - The file could be large, so we'll return it in chunks

```
@Log4j2
@Data
class Payload {

    private final byte[] bytes;
    private final int length;

    public static Payload from(byte[] bytes, int len) {
        return new Payload(bytes, len);
    }

    @Override
    public String toString() {
        return String.format("[*****THREAD %d PROCESSING PAYLOAD*****] %s",
                             Thread.currentThread().getId(),
                             new String(bytes));
    }
}
```

demo.syncasync.Payload.java

Doing Synchronous I/O (2 of 4)

- Here's an interface that specifies a read operation
  - The `read()` function reads data from the specified file, …
  - … puts data into `Payload` objects, …
  - … and passes `Payload` objects to a consumer to process

```
import java.util.function.Consumer;

public interface Reader {
    void read(String filename, Consumer<Payload> consumer);
}
```

demo.syncasync.Reader.java

- The interface doesn't specify how `read()` works
  - We'll implement the method synchronously first
  - Then we'll implement it asynchronously, to avoid wasted waiting

Doing Synchronous I/O (3 of 4)

- Synchronous implementation of the `Reader` interface
  - Makes use of `InputStream` `read()` method
  - This is blocking I/O

```
@Log4j2
@Component
@Lazy
class SynchronousReader implements Reader {

    @Override
    public void read(String filename, Consumer<Payload> consumer) {
        try (InputStream in = new FileInputStream(filename)) {
            byte[] data = new byte[1024];
            int res;
            while ((res = in.read(data, 0, data.length)) != -1) {
                consumer.accept(Payload.from(data, res));
            }
        }
        catch (IOException ex) {
            log.error(ex.getMessage());
        }
    }
}
```

demo.syncasync.SynchronousReader.java

Doing Synchronous I/O (4 of 4)

- Client code can use the synchronous reader as follows
  - Run this code and see what happens
  - When will it display "main thread doing useful work" messages?

```
@Log4j2
@SpringBootApplication
public class Application {

    public static void main(String[] args) {

        ApplicationContext context = SpringApplication.run(Application.class, args);
        Reader reader = context.getBean(SynchronousReader.class);
        doRead("Data/Macbeth.xml", reader);
    }

    private static void doRead(String filename, Reader reader) {

        reader.read(filename, bb -> System.out.println(bb));

        for (int i = 0; i < 10; i++) {
            System.out.println("[*****MAIN THREAD DOING USEFUL WORK*****]");
            try { Thread.sleep(1000); }  catch (InterruptedException ex) {}
        }
    }
}
```

demo.syncasync.Application.java

Characteristics of Synchronous I/O

- Synchronous I/O is *pull-model* processing
  - We're pulling bytes out of a data source (e.g. an `InputStream`)
- This is fine if the data source is fast
  - E.g. the local file system
- It's not fine if the data source is slow
  - E.g. a network file, or a remote service
  - When we call `in.read()`, it could take a very long time
- Running the code on a separate thread doesn't help
  - We're limited to the number of threads on our core
  - Eventually we'll run out of threads - not infinitely scalability!

Asynchronous I/O to the Rescue

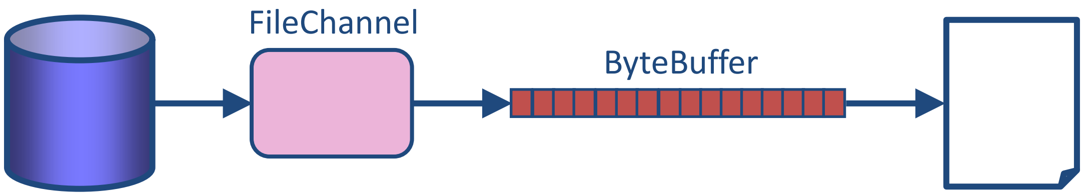
- Synchronous I/O is problematic if the data stream is slow
  - The only way to handle more I/O is to add more threads
  - But our ability to add more threads is finite

- If the bulk of your work is I/O:
  - Then asynchronous I/O can help alleviate the wastage of threads

- The next few slides we show how to do asynchronous I/O using Java NIO, in the package `java.nio`
  - Provides support for low-level I/O operations
  - Based on the concept of channels and buffers

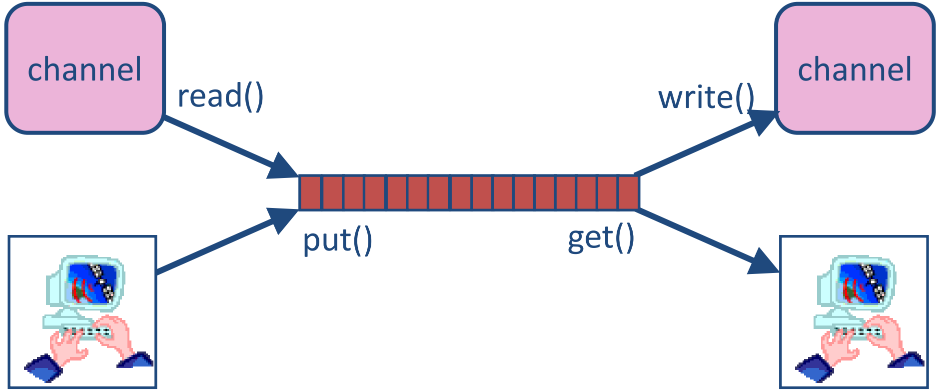
Aside: Java NIO (1 of 2)

- Channels are bidirectional sources or sinks of data
  - E.g. `FileChannel` reads/writes a file
  - Data manipulated in blocks
- Buffers are the unit of data moved through channels
  - Basic type of buffer is `ByteBuffer`, also `IntBuffer` etc.
  - Array of data
  - Operations and attributes to simplify data movement



Aside: Java NIO (2 of 2)

- To place data into a buffer
  - Invoke channel `read()` method to read data from channel
  - Or invoke buffer `put()` method to put in data manually
- To take data from a buffer
  - Invoke channel `write()` method to write data into channel
  - Or invoke buffer `get()` method to get out data manually



Doing Asynchronous I/O (1 of 3)

- Asynchronous implementation of the `Reader` interface
  - Makes use of `AsynchronousFileChannel` `read()` method
  - Non-blocking, calls back `CompletionHandler` when data ready

```
@Log4j2
@Component
@Lazy
class AsynchronousReader implements Reader, CompletionHandler<Integer, ByteBuffer> {

    private long position;
    private AsynchronousFileChannel fileChannel;
    private Consumer<Payload> consumer;

    public void read(String filename, Consumer<Payload> c) {
        this.consumer = c;
        try {
            this.fileChannel = AsynchronousFileChannel.open(
                    Paths.get(filename),
                    StandardOpenOption.READ);

            ByteBuffer buffer = ByteBuffer.allocate(1024);
            this.fileChannel.read(buffer, 0, buffer, this);
        }
        catch (IOException ex) { … }
    }
    …
```

demo.syncasync.AsynchronousReader.java

Doing Asynchronous I/O (2 of 3)

- We implement the `CompletionHandler` interface to handle data as soon as it is available

```
@Log4j2
@Component
@Lazy
class AsynchronousReader implements Reader, CompletionHandler<Integer, ByteBuffer> {
    …
    @Override
    public void completed(Integer bytesRead, ByteBuffer buffer) {

        if (bytesRead < 0)
            return;

        // Flip FileChannel buffer from write- to read-mode, then read bytes from it.
        buffer.flip();
        byte[] data = new byte[buffer.limit()];
        buffer.get(data);

        // Put the data into our Payload object, and consume (process) it.
        consumer.accept(Payload.from(data, data.length));

        // Clear the FileChannel buffer, and fire off the next read.
        buffer.clear();
        this.position = this.position + bytesRead;
        this.fileChannel.read(buffer, this.position, buffer, this);
    }
}
```

demo.syncasync.AsynchronousReader.java

Doing Asynchronous I/O (3 of 3)

- Client code can use the asynchronous reader as follows
  - Run this code and see what happens
  - When will it display "main thread doing useful work" messages?

```
@Log4j2
@SpringBootApplication
public class Application {

    public static void main(String[] args) {

        ApplicationContext context = SpringApplication.run(Application.class, args);
        Reader reader = context.getBean(AsynchronousReader.class);
        doRead("Data/Macbeth.xml", reader);
    }

    private static void doRead(String filename, Reader reader) {

        reader.read(filename, bb -> System.out.println(bb));

        for (int i = 0; i < 10; i++) {
            System.out.println("[*****MAIN THREAD DOING USEFUL WORK*****]");
            try { Thread.sleep(1000); }  catch (InterruptedException ex) {}
        }
    }
}
```

demo.syncasync.Application.java

Characteristics of Asynchronous I/O

- Asynchronous I/O is *push-model* processing
  - The `read()` operation returns on the main thread immediately
  - The main thread can do useful work in the meantime
  - When data is ready, it's pushed to our `CompletionHandler` on a separate thread

- Asynchronous I/O helps for I/O-bound operations
  - We can get better juice out of our available hardware

- Asynchronous I/O doesn't help for CPU-bound operations
  - E.g. number-crunching, etc.

2. The Promise of Reactive Programming

- From I/O to collections
- Future<T> and CompletableFuture<T>
- Iterator<T> and Stream<T>
- The essence of the problem
- Reactive programming to the rescue

From I/O to Collections

- Most coding tasks don't use `InputStream` or `Channel`, but instead tend to work with collections
  - The concepts are similar though…
  - We expect to be able to get all of the data, quite quickly

- Collections can become problematic if:
  - You're dealing with large or unbounded amounts of data
  - You're dealing with data with a lot of latency between records

- These cases require asynchrony, i.e. the ability to deal with data that will eventually arrive
  - Can `Future<T>` or `CompletableFuture<T>` help?
  - Can `Iterator<T>` or `Stream<T>` help?

Future<T> and CompletableFuture<T>

- `Future<T>` and `CompletableFuture<T>` describe a task that will eventually complete
  - But they only describe <u>one</u> completion
  - They don't describe <u>multiple</u> ongoing completions

- So they aren't a good way to represent a whole bunch of (potentially unlimited) data arrivals

Iterator<T> and Stream<T>

- `Iterator<T>` and `Stream<T>` both work fine with very large (or potentially unlimited) data streams
  - But they use a *pull model*
  - They don't push data at you when it becomes available

- If `Iterator<T>` and `Stream<T>` did have *push model* capabilities, that would raise another issue…
  - Who knows how much data the data source might push at you?
  - You'd need a way to push-back, i.e. to say "whoa tiger!"

The Essence of the Problem

- We need asynchrony
  - The ability to process data on a separate thread
- We'd like a push model
  - The data source pushes data at us, when it's available
- We need a way to push-back
  - To tell the data source, we're ready for the next xxx bytes when they're available

Reactive Programming to the Rescue

- The issues on the previous slide pertain to flow-control
  - The ability of the client to signal how much work it can handle
  - This is called <u>back pressure</u>

- Reactive programming resolves these issues
  - Several libraries available
  - E.g. RxJava, Akka Streams, Project Reactor

- Reactive Streams is an initiative that defines a standard for async stream processing with non-blocking back pressure
  - See http://www.reactive-streams.org/

3. The Usage of Reactive Programming

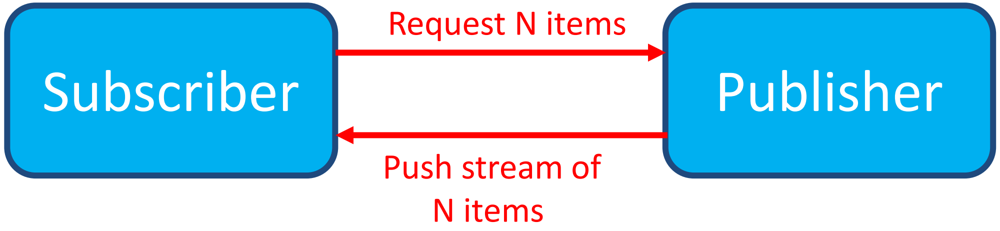
- Support for Reactive Streams in Java 9+
- Understanding the Java Flow API
- Flow API interfaces
- Implementing a subscriber class
- Main code - publishing and subscribing
- Implementing a processor class
- Subscribing to the processor class
- Main code - publishing and subscribing

Support for Reactive Streams in Java 9+

- Java 9+ supports Reactive Streams via the Flow API
  - A combination of the Iterator and Observer patterns…
- The Iterator is a *pull* model
  - The app pulls items from the source
- The Observer is a *push* model
  - Items from the source are pushed to the application

Understanding the Java Flow API

- The Java Flow API is a mixture of *pull* and *push*:
  - The subscriber initially requests N items
  - The publisher publishes at most N items to the subscriber



- Addresses the common problem of back pressure
  - Whereby buffer fills up because subscriber is too slow

Flow API Interfaces

- The Flow API defines several key interfaces inside the `java.util.concurrent.Flow` class

```
@FunctionalInterface
public static interface Flow.Publisher<T> {
    public void subscribe(Flow.Subscriber<? super T> subscriber);
}
```

```
public static interface Flow.Subscriber<T> {
    public void onSubscribe(Flow.Subscription subscription);
    public void onNext(T item) ;
    public void onError(Throwable throwable) ;
    public void onComplete();
}
```

```
public static interface Flow.Subscription {
    public void request(long n);
    public void cancel();
}
```

```
public static interface Flow.Processor<T,R>
    extends Flow.Subscriber<T>, Flow.Publisher<R> {
}
```

Implementing a Subscriber Class

```
import java.util.concurrent.Flow.*;
…
public class MySubscriber<T> implements Subscriber<T> {

    private Subscription subscription;
    public ArrayList<T> consumedItems = new ArrayList<>();

    @Override
    public void onSubscribe(Subscription subscription) {
        this.subscription = subscription;
        subscription.request(1);   // The subscription facilitates back pressure.
    }

    @Override
    public void onNext(T item) {
        System.out.println("MySubscriber onNext(): " + item);
        consumedItems.add(item);
        subscription.request(1);
    }

    @Override
    public void onError(Throwable t) {
        t.printStackTrace();
    }

    @Override
    public void onComplete() {
        System.out.println("MySubscriber onComplete()");
    }
}
```

demo.reactive1.MySubscriber.java

Main Code - Publishing and Subscribing

- Here's the main code
  - We use `SubmissionPublisher` to publish strings
  - We define a subscriber to subscribe to the flow

```
import java.util.concurrent.SubmissionPublisher;
…

// Create a publisher - we've used the SubmissionPublisher implementation class.
SubmissionPublisher<String> publisher = new SubmissionPublisher<>();

// Register a subscriber.
MySubscriber<String> subscriber = new MySubscriber<>();
publisher.subscribe(subscriber);

// Publish some items.
System.out.println("Publishing Items...");
String[] items = {"matthew", "mark", "luke", "john"};
Arrays.asList(items).stream().forEach(item -> publisher.submit(item));

// Tell subscribers we're done.
publisher.close();

System.out.printf("Subscriber consumed %d items\n", subscriber.consumedItems.size());
```

demo.reactive1.Main.java

Implementing a Processor Class (1 of 2)

- Now let's see how to implement a processor class
  - i.e. a class that implements `Flow.Processor`

```
public static interface Flow.Processor<T,R>
    extends Flow.Subscriber<T>, Flow.Publisher<R> {
}
```

- A processor class is like a *transformer*:
  - Subscribes to an upstream publisher, to receive items
  - Processes the items
  - Publishes results to a downstream subscriber

Implementing a Processor Class (2 of 2)

```
public class MyTransformProcessor<T,R>
    extends SubmissionPublisher<R>
    implements Flow.Processor<T,R> {

    private Flow.Subscription subscription;
    private Function<T,R> function;

    public MyTransformProcessor(Function<T,R> function) {
        this.function = function;
    }

    @Override
    public void onSubscribe(Flow.Subscription subscription) {
        this.subscription = subscription;
        subscription.request(1);
    }

    @Override
    public void onNext(T item) {
        R transformResult = function.apply(item);
        this.submit(transformResult);
        subscription.request(1);
    }

    @Override
    public void onError(Throwable t) { … }

    @Override
    public void onComplete() { … }
}
```

demo.reactive2.MyTransformProcessor.java

Subscribing to the Processor Class

- The processor class is part-publisher, part-subscriber
  - So we need to subscribe to its outputs

- See `MySubscriber` class in `demo.reactive2` package
  - Same code as before
  - i.e. it subscribes and accumulates results locally

Main Code - Publishing and Subscribing

- Here's the main code
  - We use `SubmissionPublisher` to publish strings
  - We use our processor to process these strings
  - We use our subscriber to subscribe to the transformed results

```
SubmissionPublisher<String> publisher = new SubmissionPublisher<>();

MyTransformProcessor<String, Integer> transformProcessor =
                                  new MyTransformProcessor<>(s -> s.length());
publisher.subscribe(transformProcessor);

MySubscriber<Integer> subscriber = new MySubscriber<>();
transformProcessor.subscribe(subscriber);

System.out.println("Publishing Items...");
String[] items = {"matthew", "mark", "luke", "john"};
Arrays.asList(items).stream().forEach(item -> publisher.submit(item));

publisher.close();

System.out.printf("Subscriber consumed %d items\n", subscriber.consumedItems.size());
```

demo.reactive2.Main.java

Summary

- The need for reactive programming
- The promise of reactive programming
- The usage of reactive programming