Modeling and Design Patterns

ISEL 2020

Week 9

Asynchronous Programming

Remember Sequences or Streams

- []
- stream (ALGOL 1965)
- list (LISP 1976)
- Iterator (C++ STL 1994)
- Iterator (Java 1.2 1998)
- IEnumerable (.net 2002)
- Stream (Java 8 2014)
- Reactive Streams (Java 9, RxJava 3, Reactor)
- Async Iterator (ES2018 and C#8 2019)
- Kotlin Flow (2019)

Sequences – MDP Outline

1. java.util.Iterable

Higher-order functions

e.g. filter(upstream, T => boolean), map(upstream, T => R), …

2. java.util.stream.Stream
   Spliterator

Internal <versus> External iteration

e.g. tryAdvance(T => void) : boolean

Async Programming

3. java.util.concurrent.Flow.Publisher and Subscriber

Reactive Streams

e.g. pub.subscribe(T => void) : void

Classification

	Multiplicity	Access		Call
T	1		`item`	

Optional<T>	1	Internal
		External
			`op.ifPresent(item -> …)`
			
			`item = op.get()`
				Blocking

Iterator<T>	*	External	`item = iter.next()`	Blocking

Spliterator<T>	*	Internal	`iter.tryAdvance(item -> …)`
			
			`iter.forEachRemaining(item -> …))`
				Blocking

CompletableFuture<T>	1	Internal	`cf.thenAccept(item -> …)`	Non-blocking
Publisher<T> (e.g. RxJava, Reactor, Kotlin Flow)	*	Internal	`pub.subscribe(item -> )`	Non-blocking

Async Iterator
(e.g C# and .Net)
	*	External	`item = await iter.next();`
			
			`for await(const item of iter) …`
				Non-blocking

Blocking (sync) versus Non-blocking (async)

```
IntStream
    .rangeClosed(1, 5)
    .forEach(System.out::println);
System.out.println("Subscribed!");
```

```
Observable
    .intervalRange(1, 5, 0, 200, TimeUnit.MILLISECONDS)
    .subscribe(System.out::println);
System.out.println("Subscribed!");
```

```
1
2
3
4
5
Subscribed!
```

```
Subscribed!
1
2
3
4
5
```

Asynchronicity

Asynchronicity allows multiple things to happen at the same time

$\Rightarrow$ Concurrently

Example: Counting the total number of lines of given files.

Async from the beginning…

1. Threads

- !!! Blocking monopolizes threads !!!!
- All threads are costly (e.g. 1Mb stack memory)
- Reading 1000 files concurrently => ~ 1Gb just for stack memory

Async from the beginning…

1. Threads
2. Tasks and thread pool

?? What is the size for the thread pool ??

Depends:

- CPU bound => Thread pool size = Number of Cores
- IO bound => Threads are being blocked waiting for the return of IO

```
Future<T> fut = ExecutorService.submit(Callable)        // Callable: () => T
Future = Container of an asynchronous result: fut.get() // get() is Blocking !!!!
```

Async from the beginning…

Synchronous Results:

- Return value
- Throw Exception

1. Threads
2. Tasks and thread pool
3. Asynchronous IO
   a) Callbaks: Should capture the two possible results: success or failure. E.g:

```
- (err) => void, (val) => void
- (Result<T>) => void
```

- (err, val) => void
- …

```
e.g: AsyncFiles.readAll(path1, (err, body) -> {…})
```

Pipeline:

API Async => API Async => API Async …

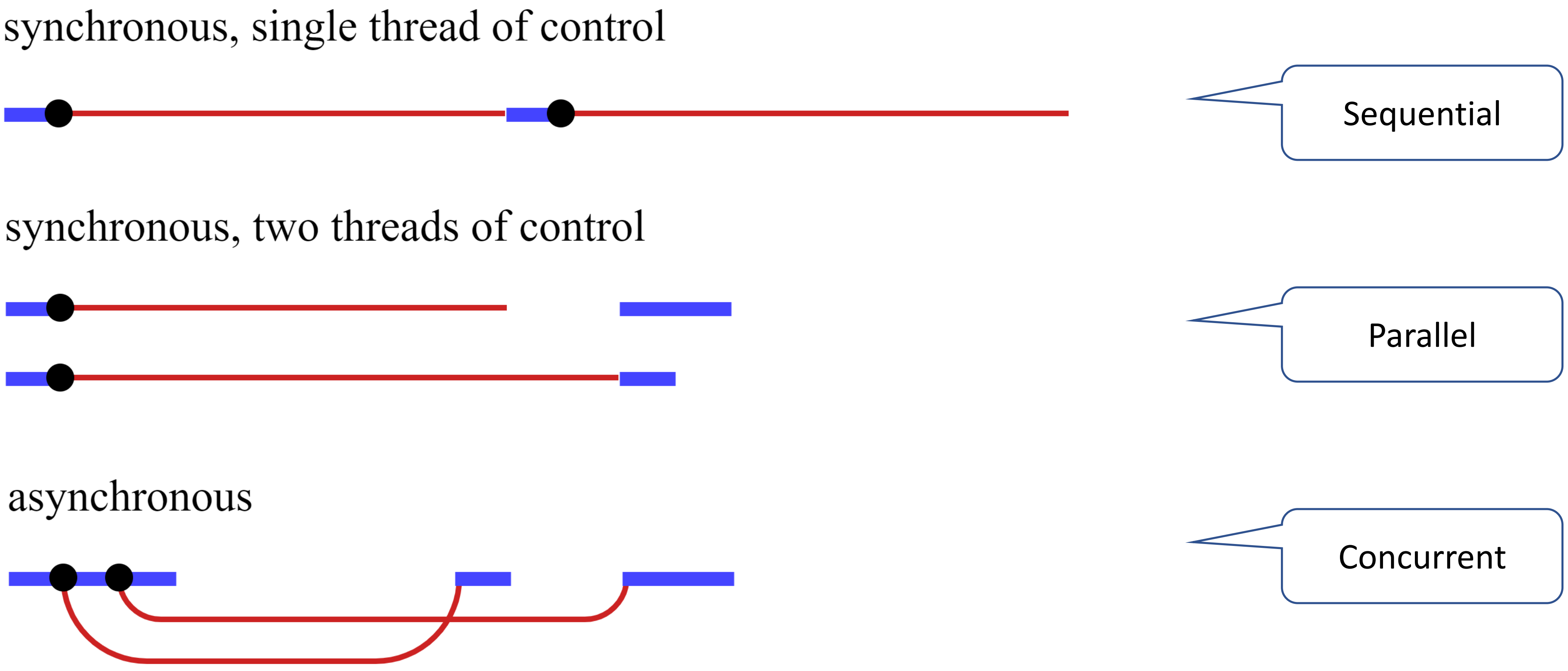
Async from the beginning…

1. Threads
2. Tasks and thread pool
3. Asynchronous IO
   a) Callbaks
   b) CompletableFuture = Future + Intermediate operations:

```
- thenApply(val ->….): CF<R>         ⇔ map( val -> ): Sream<R>
- thenCompose(val -> CF<R>): CF<R>   ⇔ flatMap(val -> Stream<R>): Strean<R>
- thenAccept(val -> …)               ⇔ forEach(val ->…): void

CompletableFuture ⇔ .Net Task ⇔ Js Promise ⇔ Deferred
```

Overview



(*) From https://eloquentjavascript.net/11_async.html

Notice

- Performance != **Throughput**
  - Performance => total execution time
  - Throughput => number of operations per second => concurrency
- Asynchronous I/O will be slower than synchronous I/O
- ↑ Throughput → more ops/sec → not exactly faster
- Processing unit => Thread
- Work unit => Task

Async API consistency

Async APIs:

- Using callbacks => provide callbacks
- Using CF => provide CF => **Composing** Advantage (pipeline)

DON’T:

- ~~Use Async API => provide Sync =>~~ e.g. `T foo() {… cf.join() …}`
- Block (except on Unit Tests)

Composing CompletableFuture

Like Streams provide pipelines:

- strm.filter(…).map(…).flatMap(…).forEach()

  op int. => op int. => op int. => op.terminal

continuations

- cf.thenApply().thenCompose(…).thenConbine(…).thenAccept()

  op int. => op int. => op int. => op.terminal

!!!! Advantage: ++ legibility ++ promote non-blocking

Aka Wikipedia:https://en.wikipedia.org/wiki/Continuation-passing_style

allOf([]Cf) ⬄ Promise.all([]Promise)

If we have been using Js:

```
var prms = ….map(AsyncFiles::readAll); // [P<String>, P<String>, …]
Promise
   .all(prms)                          // P<[String, String, ...]>
   .then(bodies => bodies.map(b => b.split(…).reduce()..))
```

But in Java we have:

```
var cfs = …
          .map(AsyncFiles::readAll); // Stream<CF<String>>
          .toArray()                 // [CF<String>, CF<String>, …]
Cf
   .allOf(cfs)   // CF<void> => completed when all CF of cfs are completed
   .thenApply((bodies) => bodies are Void ????? )
```

Goal: Services with Async API

WeatherService => WeatherWebApi => Gson

(async) (async) => HttpRequest => HTTP => world weather online

(async) AsyncHttpClient

Java asynchronous HTTP clients alternatives:

- java.net.http.HttpClient
- Apache HttpAsyncClient
- org.asynchttpclient

Goal: Services with Async API

Java asynchronous HTTP clients alternatives:

- java.net.http.HttpClient
- Apache HttpAsyncClient
- org.asynchttpclient

```java
AsyncHttpClient ahc = Dsl.asyncHttpClient();
return ahc
    .prepareGet(url)
    .execute()
    .toCompletableFuture()
    .thenApply(Response::getResponseBody)
    .thenApply(body -> ...)
```

Domain: Weather asyncchronous

AsyncWeatherApi alternatives:

?? `pastWeather(): CompletableFuture<Stream<WeatherInfo>>`

!!! Problem: Once CF is resolved then it returns always the same Stream.

=> the Stream can be 'iterated' only once.

?? pastWeather(): `CompletableFuture<Supplier<Stream<WeatherInfo>>>`

=> the Supplier could be gathered from utility `cache()`

=> !!!! Verbose !!!!

?? pastWeather(): `CompletableFuture<List<WeatherInfo>>`

=> List is caching eagerly

Better Alternative: Publisher<T> // reactive streams (later)

Reactive Streams

1. *Streams*
2. *potentially unbounded*
3. Asynchronous
4. *Nonblocking backpressure*

"composing asynchronous and event-based programs using observable collections"

Reactive Streams

1. 2009 .Net Reactive Extensions (Erik Meijer).
2. 2013 ReactiveX support for Java, Js, Scala, etc
3. 2013 Netflix, Pivotal, Lightbend, and others.
4. 2013 Alternative libraries: Akka, Play, Reactor, Vert.X, RxJava.
5. 2015 As a standard included in JDK 9 - `java.util.concurrent.Flow`.

Reactive Streams

- Observer design pattern
- Event based (e.g. `OnClickListener`)
- *publish-subscribe* (aka *observer* or *listener* (e.g. Android))