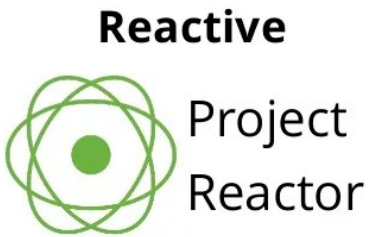
Reactive streams



- Standard interfaces: https://www.reactive-streams.org/
- Example implementation: https://projectreactor.io/ (used in Spring's WebFlux framework)
- More toolkits:
  - WebSockets, etc: https://rsocket.io/
  - Database systems: https://r2dbc.io/

Reactive streams

- Simple echo server with reactive streams:

```
DisposableServer server = TcpServer.create()
        .port(12345)
        .handle((i,o)->o.sendString(i.receive().asString()))
        .bind()
        .block();

server.onDispose().block();
```

Connect i publisher to o subscriber

Handle accept event

Wait for server socket

Main loop

Reactive streams

- Simple chat server with reactive streams:

```
Sinks.Many<String> chat = Sinks.many().multicast().onBackpressureBuffer();
chat.asFlux().subscribe(s->{});

DisposableServer server = TcpServer.create()
    .port(12345)
    .handle((i,o)->{
        i.receive().asString().subscribe(s ->
                chat.emitNext(s, Sinks.EmitFailureHandler.FAIL_FAST));
        return o.sendString(chat.asFlux());
    })
    .bind()
    .block();

server.onDispose().block();
```

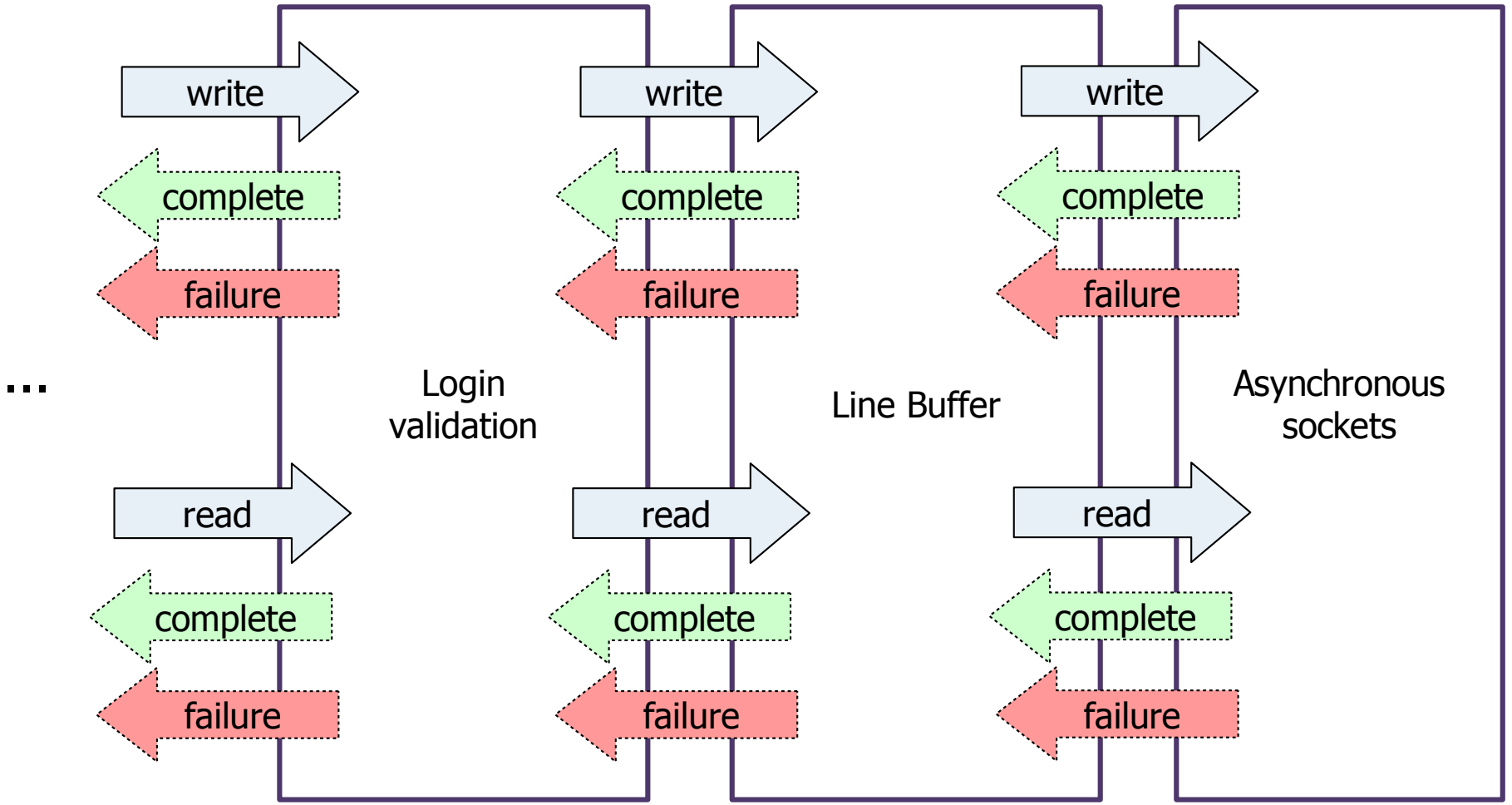
Business logic: propagate messages, but not errors/complete

Make each socket subscribe to chat

Asynchronous callbacks (again...)

- Selectors (and then reactive streams) preferred to avoid repeatedly re-issuing the same operation
- But… what about sequential work-flows that do not repeat the same operation?

Asynchronous callbacks (again...)



Asynchronous line buffer

```
public class AsynchronousLineBuffer {
    private AsynchronousSocketChannel sock;

    private CompletionHandler<String, Object> rHandler;
    private Object rValue;

    public <A> void readLine(A value, CompletionHandler<String, A> handler) {

        public void complete(...) {
            if (rHandler != null) rHandler.complete(..., rValue);
        }
    }
    private CompletionHandler<String, Object> wHandler;
    private Object wValue;
    public <A> void writeLine(String line, CompletionHandler<Void, A> handler) {
        ...
    }
}
```

Repeated code!

Asynchronous line buffer

```
public class AsynchronousLineBuffer {
      private AsynchronousSocketChannel sock;

      public BoxOf<String> readLine() {
            // call read(...) giving it an empty new BoxOf<String>()
      }

      public BoxOf<Void> writeLine(String line) {
            ...
      }
}
```

Repeated code!

Monadic asynchronous

- Encapsulate call-back in a standard reusable class: `CompletableFuture`
  - Created by the callee
  - Can be returned to the caller
  - Operations can be specified on a yet "empty box" and execute when the "box is filled up"
  - Allows synchronous waiting by threaded code

a.k.a. BoxOf<….>

- How to use:
  - Non-blocking method returns some Value
  - Blocking method returns some CompletableFuture<Value>

Translation to CompletableFuture

```
try {
      C c = codeBefore(...);

      R r = operation(...);

      codeAfter(c, r);

} catch(Exception e) {
      handleException(e);
}
```

```
C c = codeBefore(...);

asyncOperation(...)

      .thenAccept( (r) → codeAfter(c, r) )


      .exceptionally( (e) → handleException(e) )
```

Cheat Sheet

- Obtaining a future from scratch:

	Input			Output		
Operator	*none*	*now*	*code*	*none*	*value*	*exception*
`new`	x				x	x
`completedFuture`		x			x	
`failedFuture`		x				x
`runAsync`			x	x		x
`supplyAsync`			x		x	x

Cheat Sheet

- Composition with non-blocking code:

functional map(...)

	Input			Output		
Operator	*none*	*value*	*exception*	*none*	*same*	*new*
`thenRun`	x			x		
`thenAccept`		x		x		
`thenApply`		x				x
`exceptionally`			x			x
`handle`		x	x			x
`whenComplete`		x	x		x	

(*Async variants run handler in background thread)

Cheat Sheet

- Composition with blocking code:

blocking map(...)

	Parallel composition			Input		Output	
Operator	*no*	*both*	*either*	*none*	*value*	*none*	*value*
`thenCompose`	x				x		x
`thenCombine`		x			x		x
`runAfterBoth`		x		x		x	
`runAfterEither`			x	x		x	
`applyToEither`			x		x		x
`allOf`		x		x		x	
`anyOf`			x	x			x

(*Async variants run handler in background thread)

Monadic line buffer

```
public class LineBuffer {

    private SocketChannel sock;

    public  String readLine() {

        ...
        sock.read(...);
        ...
            readLine();

        return line;
    }

    public void writeLine(String line) {
        ...
    }
}
```

Recursive and blocking

Monadic asynchronous line buffer

```
public class FutureLineBuffer {

    private FutureSocketChannel sock;

    public CompletableFuture<String> readLine() {


        ...
        return sock.read(...)
            .thenCompose( (r) → { ...; return readLine(); } )
        ...

        return CompletableFuture.completed(line);
    }

    public CompletableFuture<Void> writeLine(String line) {
        ...
    }
}
```

Translation to Async/await

```
try {
      C c = codeBefore(...);

      R r = operation(...);

      codeAfter(c, r);

} catch(Exception e) {
      handleException(e);
}
```

```
try {
      C c = codeBefore(...);

      R r = await(operation(...));

      codeAfter(c, r);

} catch(Exception e) {
      handleException(e);
}
```

Must be contained in method that returns CompletableFuture<T>

Async/await

- Emphasis on:
  - Allowing imperative constructs (loops, try/catch, ...)
- Otherwise, the same as monadic asynchronous
- Example:
  - https://github.com/electronicarts/ea-async
- Exists in other languages:
  - https://docs.python.org/3/library/asyncio-task.html

Monadic asynchronous

- Emphasis on:
  - Hiding inversion of control
  - Composition with both synchronous and asynchronous code
- Threading:
  - Prefer functional code (without side-effects)
  - Safe to a threaded application with futures
- Socket wrapper with Futures:

https://github.com/spullara/java-future-jdk8

Event-driven programming

- Two basic event-driven approaches:
  - Callbacks and Selectors
- Application structuring and composition with:
  - Monadic futures; Reactive streams; and Async/await
- Compared to multi-threaded code:
  - Recognize the equivalence!
  - Better fit for different programs but…

  …modern frameworks close the gap