Challenges

- Reuse and composition:
  - Similar to BufferedReader? ObjectInputStream?
- Initial simplification:
  - Consider only incoming data (accept and read)

Polled I/O in Java

- Remember the main loop…
  - Mostly generic code
  - The application defines what to do with received data
- Define an interface between generic and application specific code

Generic main loop

```
if (key.isReadable()) {
    ByteBuffer buf=ByteBuffer.allocate(…);
    SocketChannel s=(SocketChannel)key.channel();

    try {
        int r=s.read(buf);
        if (r>0) {
            buf.flip();
            …
        } else {
            key.cancel();
            s.close();
            …
        }
    } catch(Exception e) { … }
}
```

New data available

Complete

Error

Generic main loop

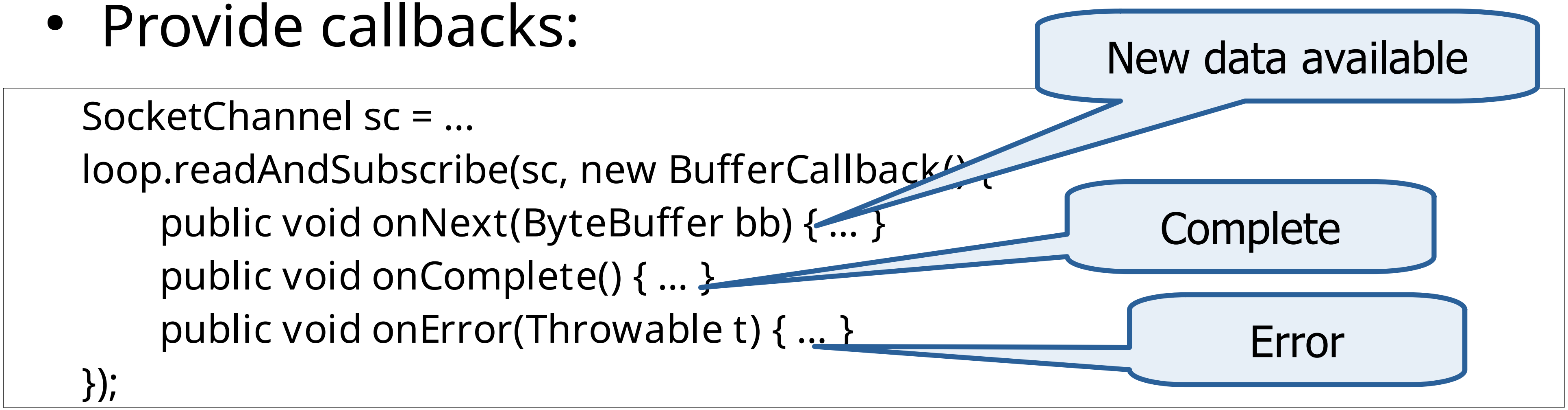
```
if (key.isReadable()) {
        ByteBuffer buf=ByteBuffer.allocate(…);
        SocketChannel s=(SocketChannel)key.channel();
        BufferCallback cb=(BufferCallback)key.attachment();
        try {
                int r=s.read(buf);
                if (r>0) {
                        buf.flip();
                        cb.onNext(buf);
                } else {
                        key.cancel();
                        s.close();
                        cb.onComplete();
                }
        } catch(Exception e) { cb.onError(e); }
}
```

Generic main loop

- Encapsulate generic code:

```
MainLoop mainloop = new MainLoop();
mainloop.run();
```

- Provide callbacks:



```
SocketChannel sc = …
loop.readAndSubscribe(sc, new BufferCallback() {
        public void onNext(ByteBuffer bb) { … }
        public void onComplete() { … }
        public void onError(Throwable t) { … }
});
```

New data available

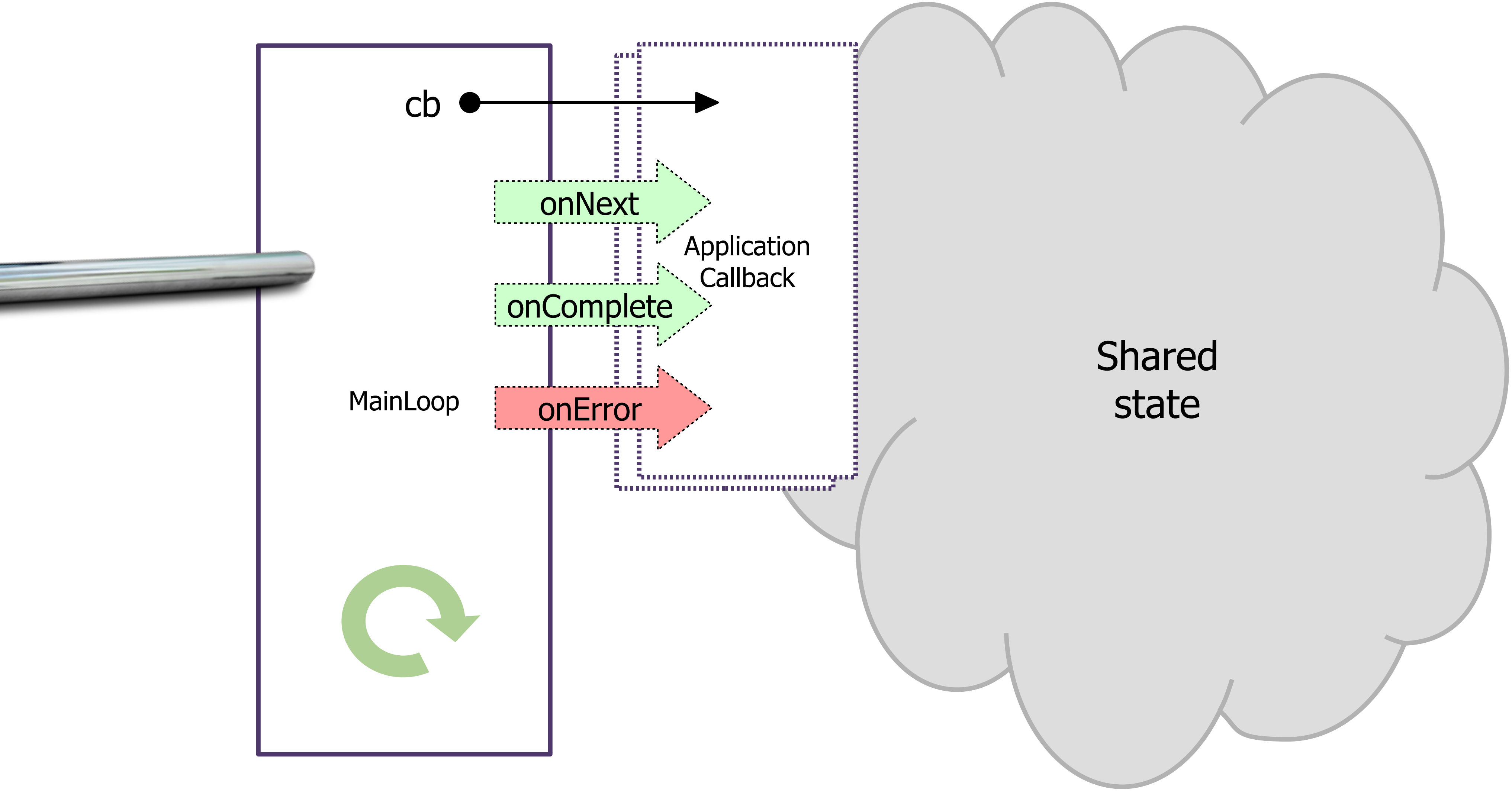
Complete

Error

Generic main loop

```
public class Mainloop {
	public void readAndSubscribe(SocketChannel s, BufferCallback cb) {
		s.configureBlocking(false);
		s.register(sel, SelectionKey.OP_READ, cb);
	}

	…
}
```

Server architecture



Layers

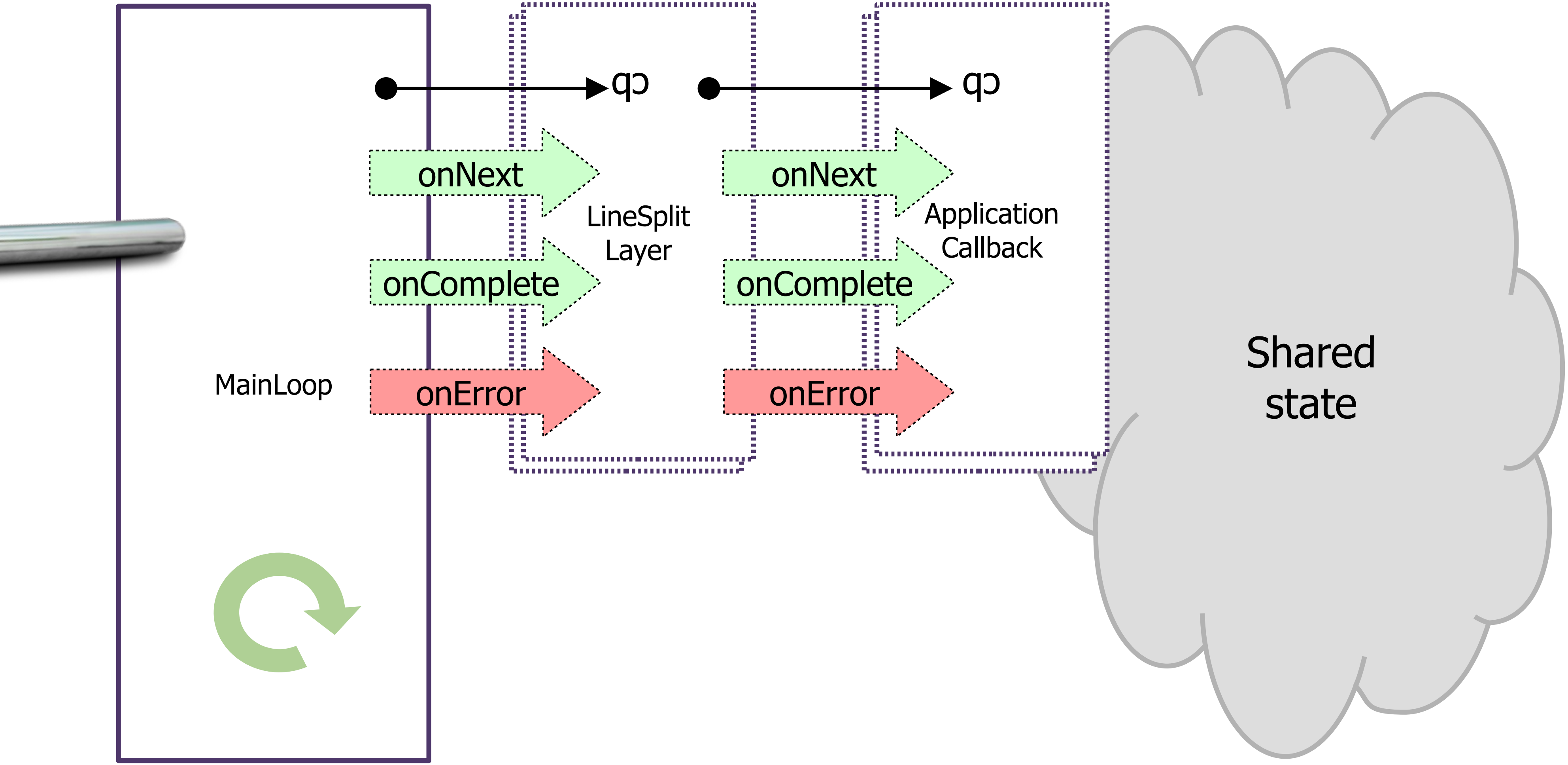
- Now we can use the callback interface to define additional layers between the main loop and the application
- Example: Split lines

Layers

```
public class LineSplitLayer implements BufferCallback {
	private BufferCallback cb;
	public void subscribe(BufferCallback cb) { this.cb = cb; }

	public void onNext(ByteBuffer bb) {
		while(bb.hasRemaining()) {
			byte b = bb.get(); line.put(b);
			if (b == ‘\n’ || !line.hasRemaining()) {
				line.flip();
				cb.onNext(line);
				line = ByteBuffer.allocate(…);
			}
		}
	}
	public void onComplete() { … cb.onComplete(); }
	public void onError(Throwable t) { …  cb.onError();}
});
```

Common to all layers

Buffer-based application



Layers

- Set up stack and callbacks:

```
SocketChannel sc = …

LineSplitLayer lines = new LineSplitLayer();
loop.readAndSubscribe(sc, lines);

lines.subscribe(new Callback() {
        public void onNext(ByteBuffer bb) { … }
        public void onComplete() { … }
        public void onError(Throwable t) { … }
});
```

What if first data arrives here?

Challenges

- How to start only after the pipeline is ready?
- …and how to stop it when done?
  - Application notifies line buffer layer
  - Line buffer layer notifies main loop
    - Removes OP_READ interest
- Changes needed:
  - A back reference
  - Updated upon subscription

Challenges

- The line split layer should produce strings…
- Changes needed:
  - A StringCallback
- Consider a Filter layer.
  - What callback interface should it implement?
  - String or ByteBuffer?

Reactive streams

- Have a generic callback interface for a stream of objects of type T (sink):
  - Observer<T>
- Have a generic utility class for managing subscriptions to a stream (source):
  - Observable<T>
- Have generic operators that are implement both Observer<T> and Observable<R>

Reactive streams

- Simple example:

```
Observable.just("a", "b", "c")
        .subscribe(m->{
            System.out.println("received "+m);
        });
```

onNext() method

- Asynchronous observable and cancelation:

```
var d = Observable.interval(1, TimeUnit.SECONDS)
        .subscribe(i->System.out.println(i));
Thread.sleep(10000);
d.dispose();
```

Implementing observables

- An observable can be implemented by:
  - Handling the initial subscription to initialize the stream
  - Calling back onNext(), … when appropriate

Implementing observables

```
public class Mainloop {
        public Observable<ByteBuffer> read(SocketChannel s) {
                return Observable.create(sub -> {
                        s.configureBlocking(false);
                        s.register(sel, SelectionKey.OP_READ, sub);
                }
        }

        public run() {
                …
                if (key.isReadable()) {
                        var sub = (ObservableEmitter<ByteBuffer>) k.attachment();
                        …
                        sub.onNext(bb);
                …
        }
}
```

Implementing observables

```
public class Mainloop {
        public Observable<SocketChannel> accept(SocketChannel s) {
                return Observable.create(sub -> {
                        s.configureBlocking(false);
                        s.register(sel, SelectionKey.OP_ACCEPT, sub);
                }
        }

        public run() {
                ...
                if (key.isAcceptable()) {
                        var sub = (ObservableEmitter<SocketChannel>) k.attachment();
                        ...
                        sub.onNext(s);
                ...
        }
}
```

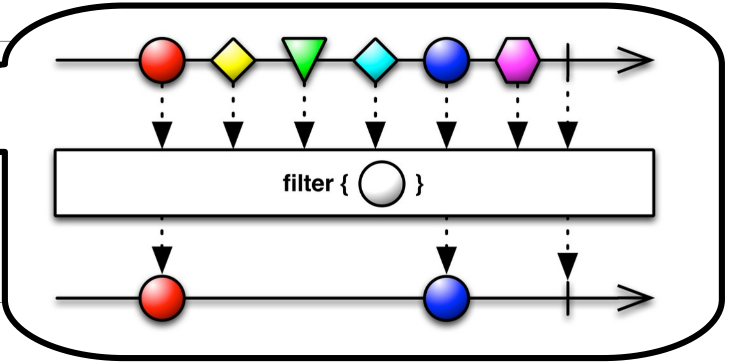
Reactive main loop

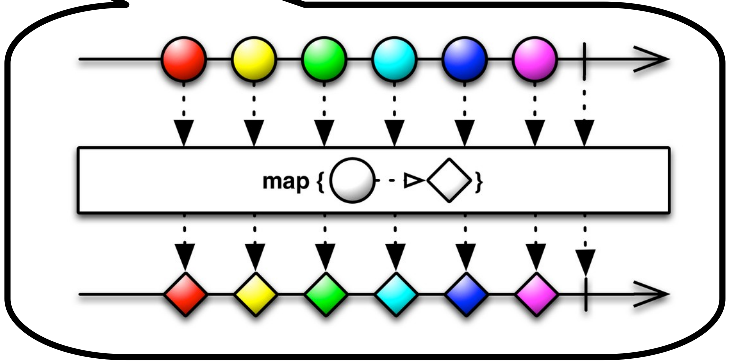
```
public class Server {
        public static void main(String[] args) throws Exception {
                var ssc = ServerSocketChannel.open(new InetSocketAddress(12345));
                var loop = new MainLoop();
                var server = loop.accept(ssc);

                server.subscribe(conn -> {
                        var obs = loop.read(conn);
                        obs.subscribe(bb -> System.out.println("received: "+bb.remaining()));
                });
        }
}
```

Functional composition

- Generic operators (and “marble diagrams”):

```
Observable.interval(1, TimeUnit.SECONDS)
        .filter(i -> i%2!=0)
        .map(i -> "received: "+i)
        .subscribe(i->System.out.println(i));
```





Custom operator

```
public class LineSplitOperator implements ObservableOperator<ByteBuffer,ByteBuffer> {

    public Observer<...> apply(Observer<...> child) throws Throwable {
        return new Observer<ByteBuffer>() {
            public void onNext(ByteBuffer bb) {
                ...
                child.onNext(...);
            }
            public void onError(Throwable e) {
                ...
               child.onError(e);
            }
            public void onComplete() {
                ...
               child.onComplete();
           }
     }
}
```

Custom operator

```
public class Server {
        public static void main(String[] args) throws Exception {
                var ssc = ServerSocketChannel.open(new InetSocketAddress(12345));
                var loop = new MainLoop();
                var server = loop.accept(ssc);

                server.subscribe(conn -> {
                        conn
                                .lift(new LineSplitOperator())
                                .map(bb-> StandardCharsets.UTF_8.decode(bb))
                                .filter(s -> !s.contains(“xxx”)
                                .subscribe(s -> System.out.println("received: "+s));
                });
        }
}
```

Reactive streams

- Implementation: https://reactivex.io/

```
<dependency>
   <groupId>io.reactivex.rxjava3</groupId>
   <artifactId>rxjava</artifactId>
   <version>3.1.6</version>
</dependency>
```

- More toolkits:
  - WebSockets, etc: https://rsocket.io/
  - Database systems: https://r2dbc.io/

References

- ReactiveX / RxJava documentation: https://reactivex.io/
- RXMarbles: https://rxmarbles.com/
- Tomasz Nurkiewicz and Ben Christensen. *Reactive Programming with RxJava: Creating Asynchronous, Event-Based Applications.* O'Reilly, 2017.
  - Chaps. 1-3