Asynchronous line buffer

```
public class AsynchronousLineBuffer {

    private AsynchronousSocketChannel sock;

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

    public <A> void readLine(final 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!

Monadic asynchronous

- Encapsulate call-back in a standard reusable class: `CompletableFuture`
  - Created by the callee
  - Can be returned to the caller
  - Allows cancellation and multiple call-backs
  - Allows synchronous waiting (future)
- How to use:
  - Non-blocking method returns some Value
  - Blocking method returns some CompletableFuture<Value>

Monadic asynchronous

- Provide composition of call-back instances
  - Chain non-blocking code: `thenApply()`
  - Chain blocking code: `thenCompose()`
- Many other combinators:
  - CompletableFuture.allOf(…) to execute multiple concurrent activities
- Long lived methods:
  - Use Async version of methods for-background thread

Monadic line buffer

```
public class LineBuffer {

      private SocketChannel sock;

      public  String readLine() {


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

            return line;
      }

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

Monadic asynchronous line buffer

```
public class FutureLineBuffer {

    private FutureSocketChannel sock;

    public CompletableFuture<String> readLine() {


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

        return CompletableFuture.completed(line);
    }

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

Monadic line buffer

- Application code is similar to single threaded sequential code:

```
try {
      String line = buf.read();
      buf.write(line);
      System.out.println(“Done”);
} catch(Exception e) {
      e.printStackTrace();
}
```

```
      c = buf.read()
       .thenCompose((l) -> buf.write(l))
       .thenRun(() -> System.out.println(“Done”))

       .exceptionally((e) -> e.printStackTrace());

      c.get();  /* synchronous wait for completion */
```

Monadic asynchronous

- Emphasis on:
  - Hiding inversion of control
  - Composition with both synchronous and asynchronous code
- Threading:
  - Prefer functional code (without side-effects)
  - Safe to the application with futures
- Example:

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

Event-driven code

- Three event-driven approaches:
  - Callbacks
  - Monadic futures
- Compared to multi-threaded code:
  - Can do the same thing
  - Better fit for different programs

More...

- Reducing memory (and synchronization) overhead:
  - LMAX Disruptor (see discussion)
  - Cap'n'Proto

- Event-driven frameworks: