Java 11 Reactive HTTP Client

Chris Hegarty (@chegar999)
Consulting Member of Technical Staff
Java Platform Group
Oracle
JPoint, 2019

JPoint
Международная Java-конференция

JavaYourNext
(Cloud)

Chris Hegarty (@chegar999)

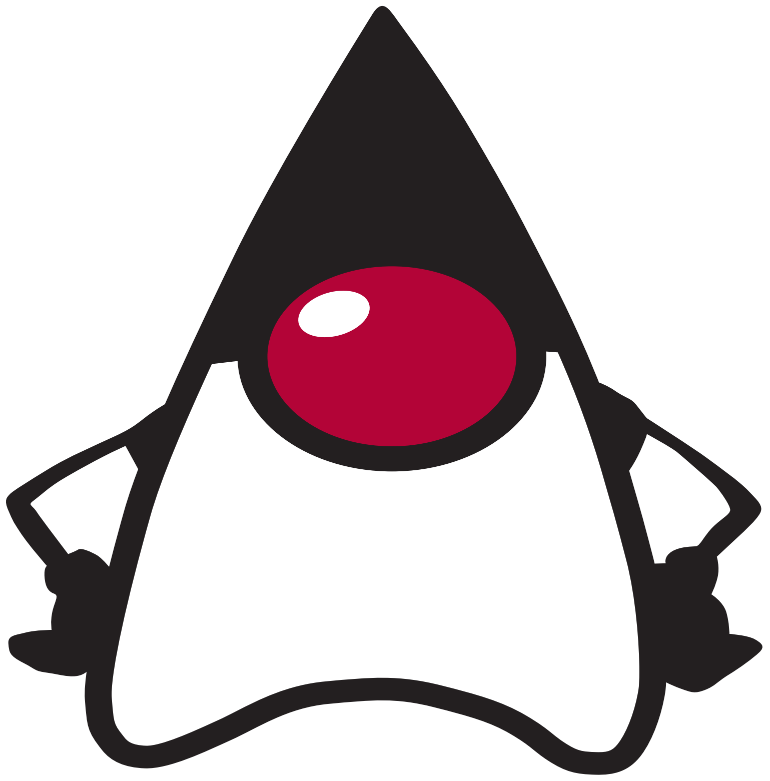
- Dublin City University
- Sun Microsystems & Oracle, since 2000
- Java Platforms and the JDK

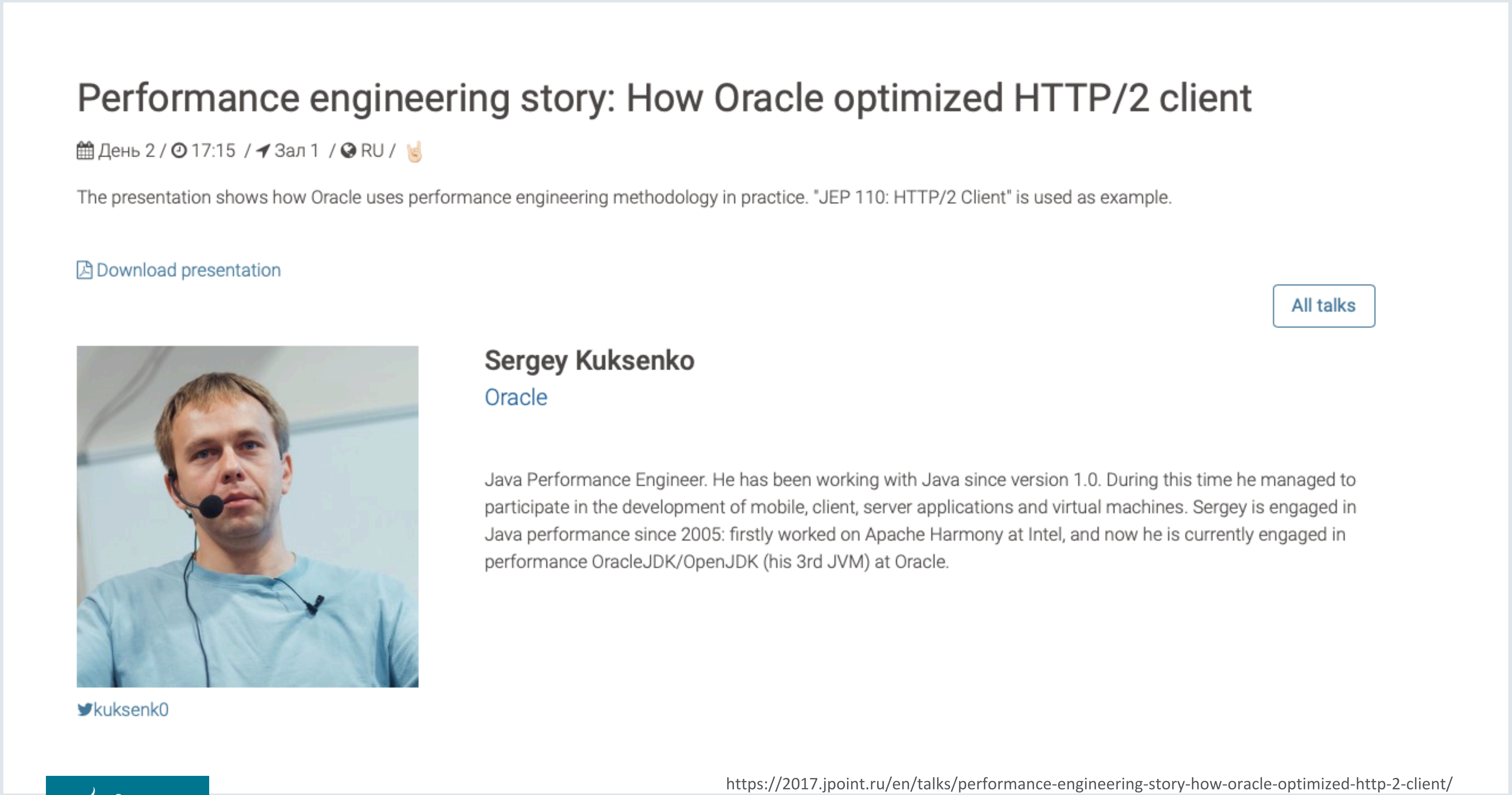
https://en.wikipedia.org/wiki/Ireland
https://en.wikipedia.org/wiki/Guinness
https://en.wikipedia.org/wiki/Saint_Patrick%27s_Day

Safe Harbor Statement

Java 11 Reactive HTTP Client

1. **Road to the Java HTTP Client**
2. **API Introduction**
3. **Handling request/response data**





Performance engineering story: How Oracle optimized HTTP/2 client

День 2 / 17:15 / Зал 1 / RU /

The presentation shows how Oracle uses performance engineering methodology in practice. "JEP 110: HTTP/2 Client" is used as example.

Sergey Kuksenko

Oracle

Java Performance Engineer. He has been working with Java since version 1.0. During this time he managed to participate in the development of mobile, client, server applications and virtual machines. Sergey is engaged in Java performance since 2005: firstly worked on Apache Harmony at Intel, and now he is currently engaged in performance OracleJDK/OpenJDK (his 3rd JVM) at Oracle.

kuksenk0

https://2017.jpoint.ru/en/talks/performance-engineering-story-how-oracle-optimized-http-2-client/

Common Q's?

1. **Why are we doing this?**
2. **How long does it take?**
3. **What are the benefits?**

~~`HttpURLConnection`~~

Motivation from JEP 110

The existing `HttpURLConnection` API and its implementation have numerous problems:

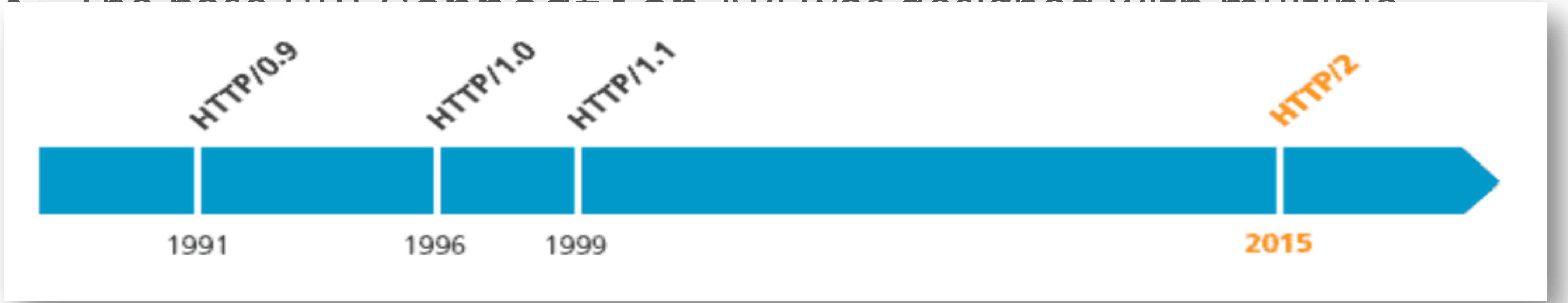
- The base `URLConnection` API was designed with multiple protocols in mind, much of which are now defunct (netdoc, gopher, etc.).
- The API predates *HTTP/1.1* and is too abstract.

* JEP - JDK Enhancement-Proposal

Motivation from JEP 110

The existing `HttpURLConnection` API and its implementation have numerous problems:

- The base `URLConnection` API was designed with multiple



* JEP - JDK Enhancement-Proposal

Motivation from JEP 110

The existing `HttpURLConnection` API and its implementation have numerous problems:

- The base `URLConnection` API was designed with multiple protocols in mind, much of which are now defunct (netdoc, gopher, etc.).
- The API predates *HTTP/1.1* and is too abstract.

* JEP - JDK Enhancement-Proposal

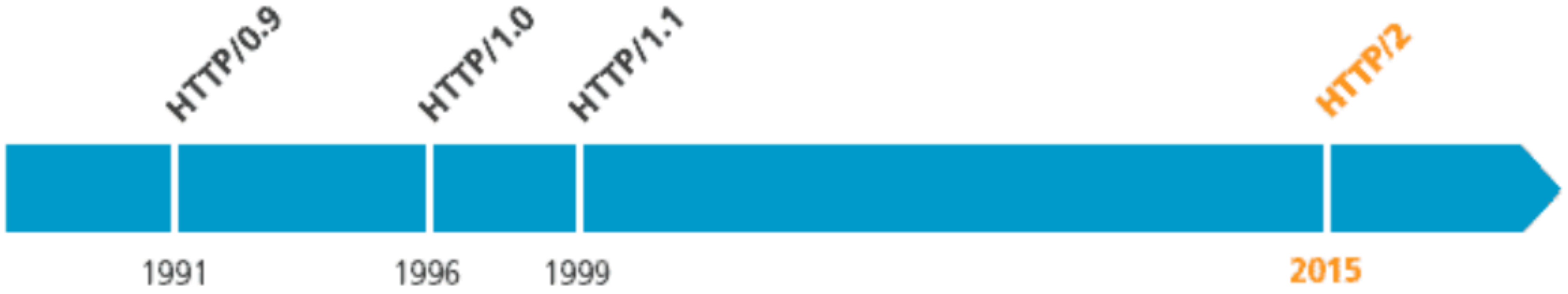
Motivation from JEP 110

The existing `HttpURLConnection` API and its implementation have numerous problems:

- The base `URLConnection` API was designed with multiple protocols in mind, much of which are now defunct (netdoc, gopher, etc.).
- The API predates *HTTP/1.1* and is too abstract
- It is hard to use, and has many undocumented behaviors
- It works in blocking mode only (i.e., one thread per request/ response).

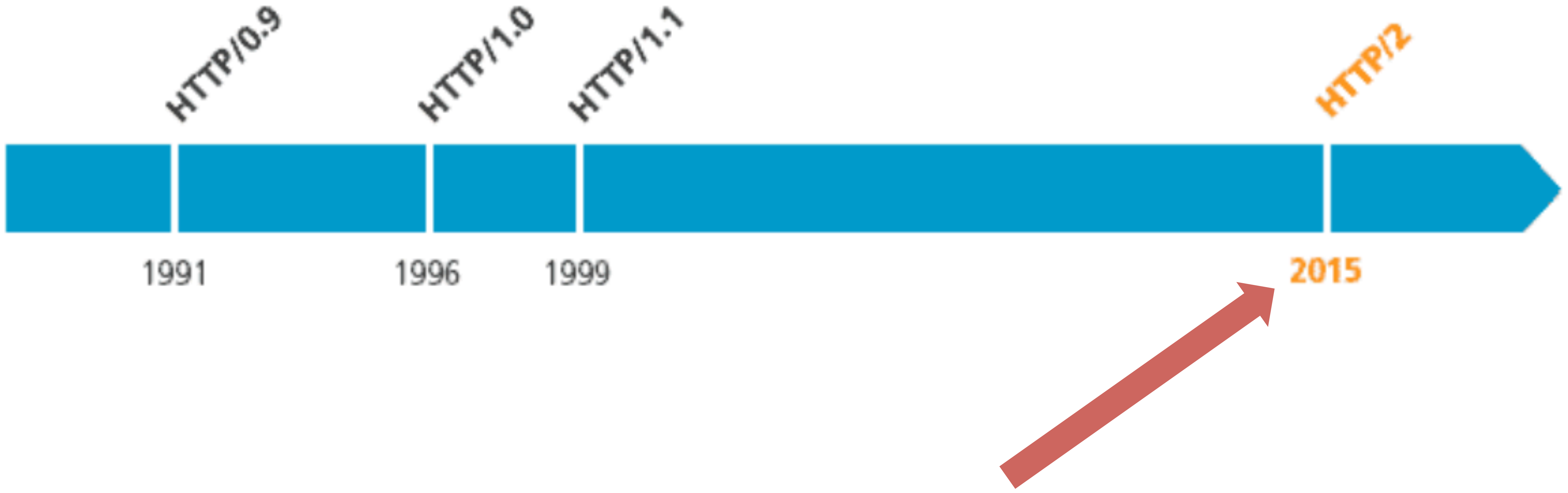
* JEP - JDK Enhancement-Proposal

Motivation from JEP 110



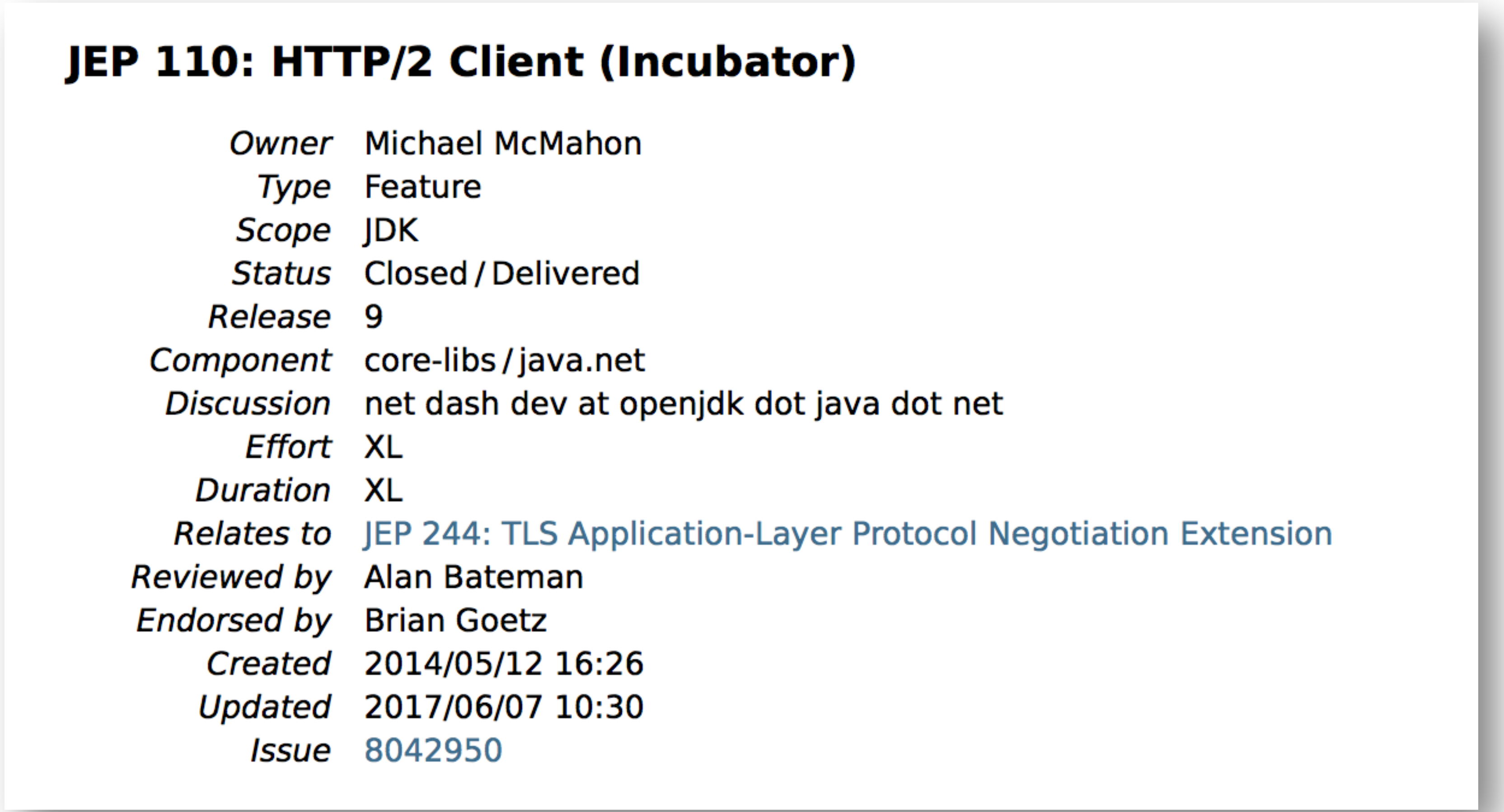
* JEP - JDK Enhancement-Proposal

Motivation from JEP 110



* JEP - JDK Enhancement-Proposal

Incubation



JEP 110: HTTP/2 Client (Incubator)

Owner	Michael McMahon
Type	Feature
Scope	JDK
Status	Closed / Delivered
Release	9
Component	core-libs / java.net
Discussion	net dash dev at openjdk dot java dot net
Effort	XL
Duration	XL
Relates to	JEP 244: TLS Application-Layer Protocol Negotiation Extension
Reviewed by	Alan Bateman
Endorsed by	Brian Goetz
Created	2014/05/12 16:26
Updated	2017/06/07 10:30
Issue	8042950

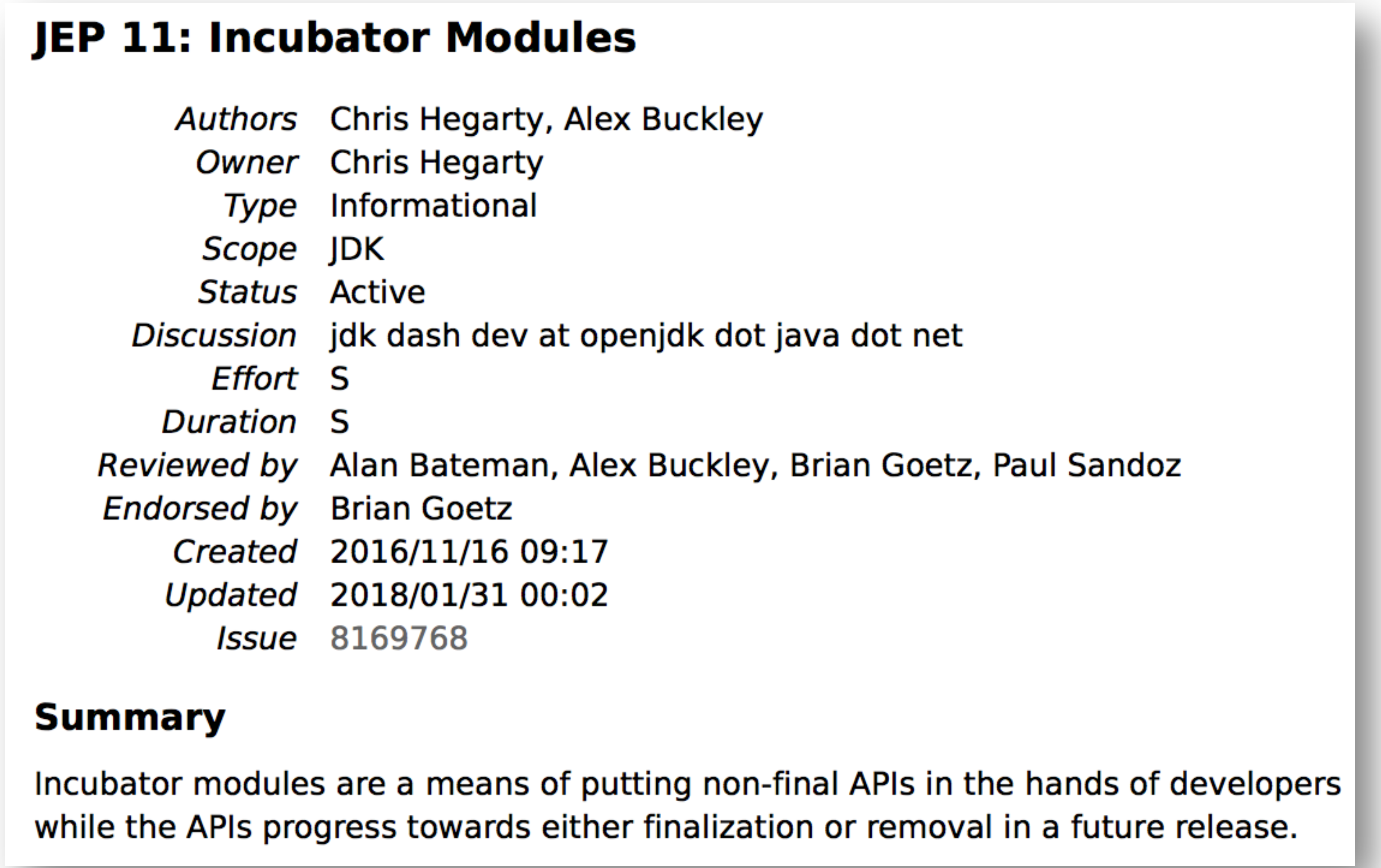
* JEP - JDK Enhancement-Proposal

Incubation

- JEP 110 was integrated in JDK 9.
- JDK 9 shipped Sep 2017
- "Incubating Feature”
    - Module `jdk.incubator.httpclient`
    - Package `jdk.incubator.http`
    - `Class jdk.incubator.http.HttpClient`

* JEP - JDK Enhancement-Proposal

Incubation



JEP 11: Incubator Modules

Authors	Chris Hegarty, Alex Buckley
Owner	Chris Hegarty
Type	Informational
Scope	JDK
Status	Active
Discussion	jdk dash dev at openjdk dot java dot net
Effort	S
Duration	S
Reviewed by	Alan Bateman, Alex Buckley, Brian Goetz, Paul Sandoz
Endorsed by	Brian Goetz
Created	2016/11/16 09:17
Updated	2018/01/31 00:02
Issue	8169768

Summary

Incubator modules are a means of putting non-final APIs in the hands of developers while the APIs progress towards either finalization or removal in a future release.

* JEP - JDK Enhancement-Proposal

Incubation

- JEP 110 was integrated in JDK 9.
- JDK 9 shipped Sep 2017
- "Incubating Feature”
  - Module `jdk.incubator.httpclient`
  - Package `jdk.incubator.http`
  - `Class jdk.incubator.http.HttpClient`

* JEP - JDK Enhancement-Proposal

Incubation

- JEP 110 was integrated in JDK 9.
- JDK 9 shipped Sep 2017
- "Incubating Feature”
  - Module `jdk.incubator.httpclient`
  - Package `jdk.incubator.http`
  - `Class jdk.incubator.http.HttpClient`
- Refreshed in JDK 10
  - Improved and more robust implementation
  - API changes, developer feedback and additional experience

* JEP - JDK Enhancement-Proposal

Standardization

- Standardized in Java 11, JEP 321
  - Module `java.net.http`
  - Package `java.net.http`
  - `Class java.net.http.HttpClient`
- Incubating version completely REMOVED

* JEP - JDK Enhancement-Proposal

Benefits

JEP 110 Goals:

- Support HTTP/2
- Provide Asynchronous API
- Modernized API (using newer Java APIs and language features)
- Support WebSocket handshake
- …

* JEP - JDK Enhancement-Proposal

Benefits: HTTP/2

- Header Compression. HTTP/2 uses HPACK compression, which reduces overhead.
- Single Connection to the server, reduces the number of round trips needed to set up multiple TCP connections.
- Multiplexing. Multiple requests are allowed at the same time, on the same connection.
- Server Push. Additional future needed resources can be sent to a client.
- Binary format. More compact.

Benefits: HTTP/2

Version support in the Java HTTP Client:

- HTTP/1.1 and HTTP/2
- Prefers HTTP/2, by default
  - Tries to Upgrade clear text requests
  - Tries to negotiate h2 in the ALPN for HTTP over TLS
    (TLS 1.3 support, leverages from JDK 11)

Benefits: Modernization

- Asynchronous API
  - `java.util.concurrent.CompletableFuture`
- Follows familiar builder style
- Immutable types
- Reactive-Streams based body processing
  - `java.util.concurrent.Flow.[Subscriber|Publisher]`

1. Where to find the API?
2. How to send a request?
3. How to handle request/response body?

HttpClient

To send a request, first create an `HttpClient` from its builder.
The builder can be used to configure per-client state, like:

- The preferred protocol version (HTTP/1.1 or HTTP/2)
- Whether to follow redirects
- A proxy
- An authenticator
- A connect timeout
- …

HttpClient

```
HttpClient client = HttpClient.newBuilder()
  .version(Version.HTTP_2)
  .followRedirects(Redirect.NORMAL)
  .proxy(ProxySelector.of(new InetSocketAddress(”proxy”,80)))
  .authenticator(Authenticator.getDefault())
  .connectTimeout(Duration.ofSeconds(20))
  .build();
```

Once built, an `HttpClient` can be used to send multiple requests.

HttpRequest

An `HttpRequest` is created from its builder.
The request builder can be used to set:

- the request URI
- the request method (GET, PUT, POST)
- the request body (if any)
- a request timeout
- request headers

HttpRequest

```
HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create("http://openjdk.java.net/"))
    .timeout(Duration.ofMinutes(1))
    .header("Content-Type", "application/json")
    .POST(BodyPublishers.ofFile(Paths.get("file.json")))
    .build()
```

- Once built an `HttpRequest` is immutable, and can be sent multiple times.

HttpResponse<T>

```
interface HttpResponse<T> {
    int statusCode();
    HttpHeaders headers();
    T body();
    Version version();
    HttpRequest request();
    …
}
```

- An `HttpResponse` is not created directly, but rather returned as a result of sending an `HttpRequest`.

Synchronous or Asynchronous

- Requests can be sent either synchronously or asynchronously.
- The synchronous API, as expected, blocks until the `HttpResponse<#Body_Type#>` is available.
- The asynchronous API returns a `CompletableFuture<HttpResponse<#Body_Type#>>`

Synchronous

```
HttpResponse<String> response =
    client.send(request, BodyHandlers.ofString());
System.out.println(response.statusCode());
System.out.println(response.body());
```

- `BodyHandlers.ofString()` is a factory that creates body handler that accumulates the response body bytes and returns them as a `String`.

Asynchronous

```
client.sendAsync(request, BodyHandlers.ofString())
    .thenApply(response -> {
          System.out.println(response.statusCode());
          return response; } )
    .thenApply(HttpResponse::body)
    .thenAccept(System.out::println);
```

- The asynchronous API returns immediately with a `CompletableFuture` that completes with the `HttpResponse` when it becomes available. `CompletableFuture` was added in Java 8 and supports composable asynchronous programming.

Asynchronous

```
client.sendAsync(request, BodyHandlers.ofString())
    .thenApply(response -> {
          System.out.println(response.statusCode());
          return response; } )
    .thenApply(HttpResponse::body)
    .thenAccept(System.out::println);
```

- The asynchronous API returns immediately with a `CompletableFuture` that completes with the `HttpResponse` when it becomes available. `CompletableFuture` was added in Java 8 and supports composable asynchronous programming.

Asynchronous

```
client.sendAsync(request, BodyHandlers.ofString())
    .thenApply(response -> {
          System.out.println(response.statusCode());
          return response; } )
    .thenApply(HttpResponse::body)
    .thenAccept(System.out::println);
```

- The asynchronous API returns immediately with a `CompletableFuture` that completes with the `HttpResponse` when it becomes available. `CompletableFuture` was added in Java 8 and supports composable asynchronous programming.

Asynchronous

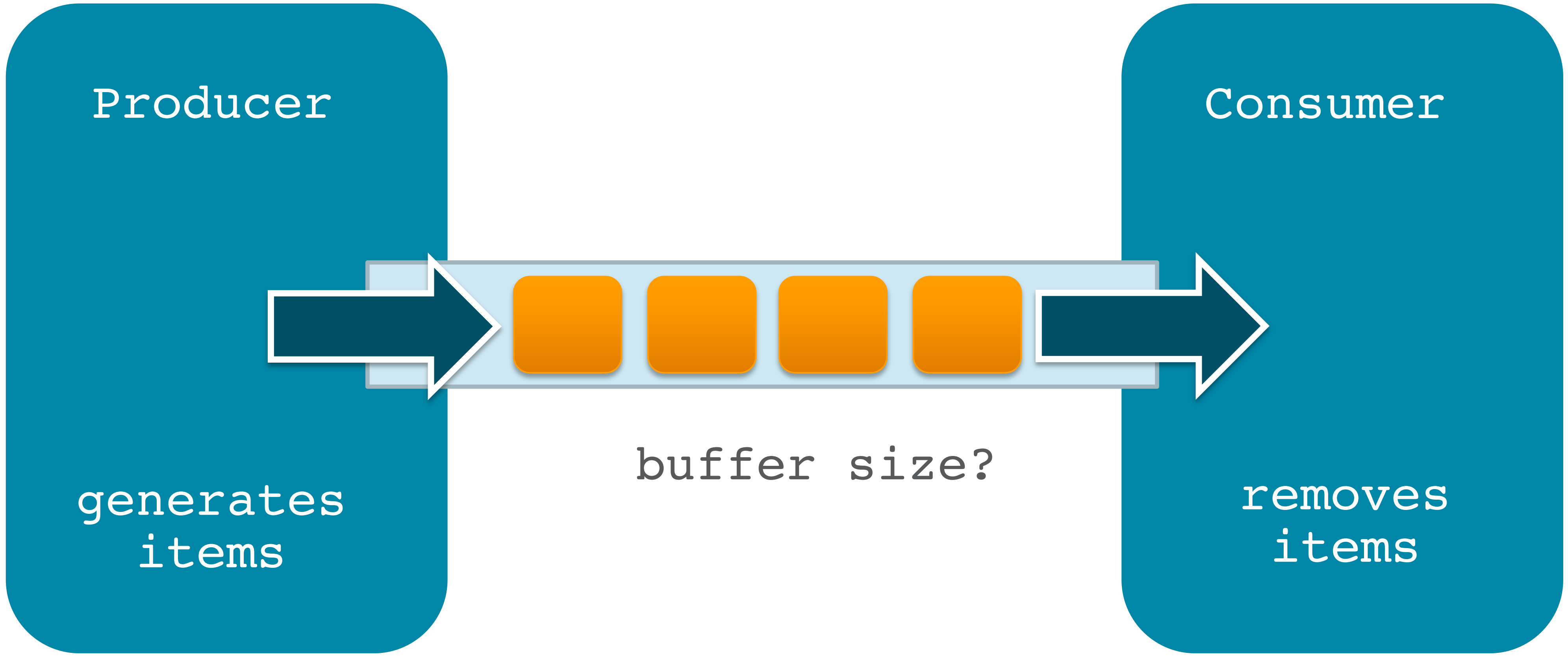
```
client.sendAsync(request, BodyHandlers.ofString())
    .thenApply(response -> {
          System.out.println(response.statusCode());
          return response; } )
    .thenApply(HttpResponse::body)
    .thenAccept(System.out::println);
```

- The asynchronous API returns immediately with a `CompletableFuture` that completes with the `HttpResponse` when it becomes available. `CompletableFuture` was added in Java 8 and supports composable asynchronous programming.

Data as reactive-streams

- The request and response bodies are exposed as reactive streams (asynchronous streams of data with non-blocking back pressure.)
- The `HttpClient` is effectively a `Subscriber` of request body and a `Publisher` of response body bytes.

Classic Producer-Consumer



Reactive-streams (40th foot view)

Producer
generates items
Consumer
removes items

Reactive-streams (40th foot view)

Publisher
publishes items
Consumer
removes items

Reactive-streams (40th foot view)

Publisher
publishes items
Subscriber
receives items

Reactive-streams (40th foot view)

Publisher
subscribe(subscriber)
publishes items
Subscriber
receives items

Reactive-streams (40th foot view)

Publisher

`subscribe(subscriber)`

publishes items

Subscriber

`onSubscribe(sub)`

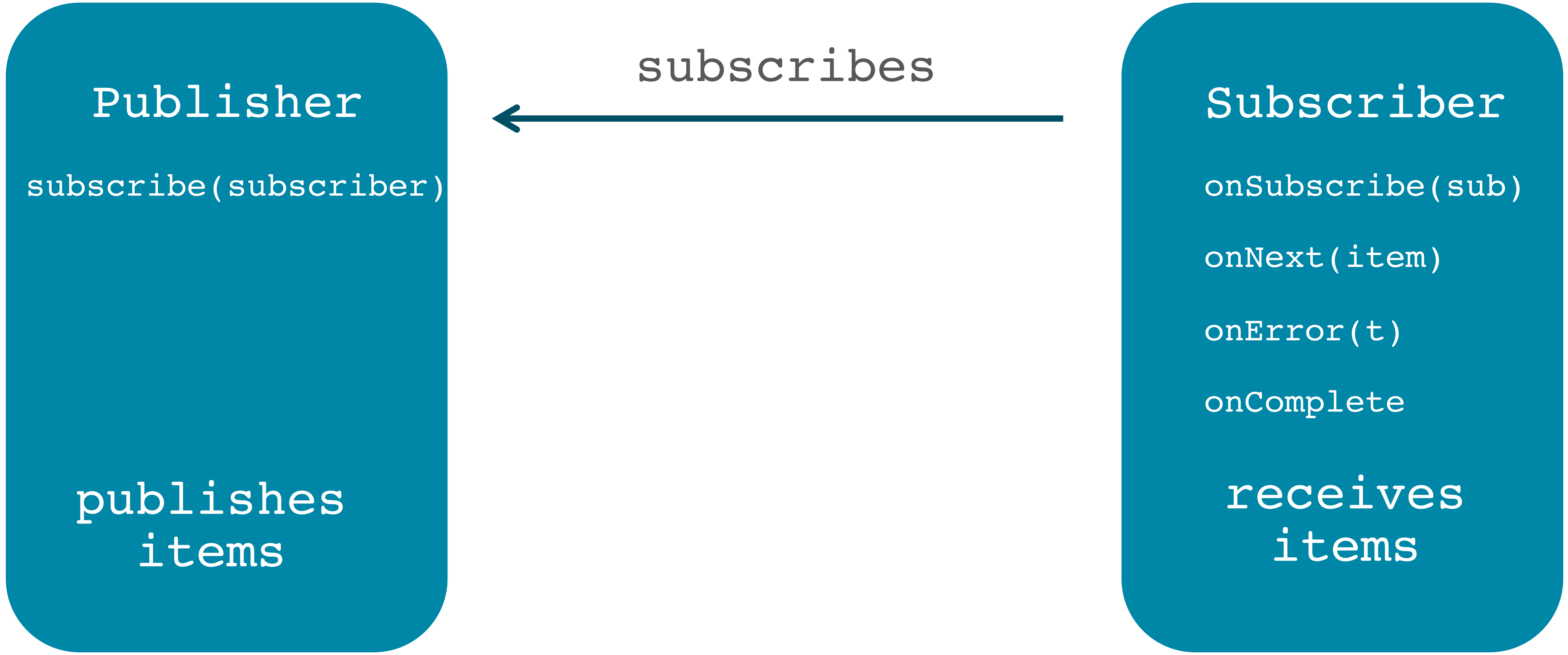
`onNext(item)`

`onError(t)`

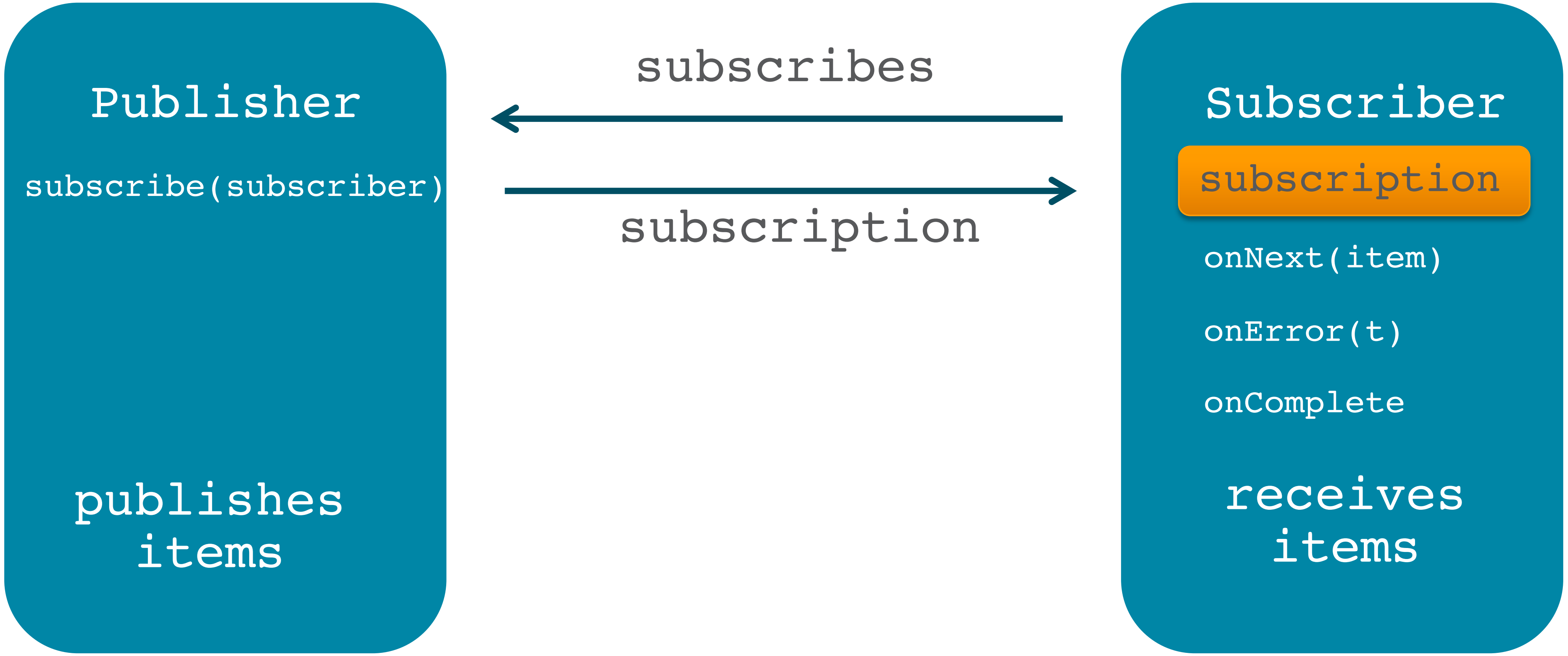
`onComplete`

receives items

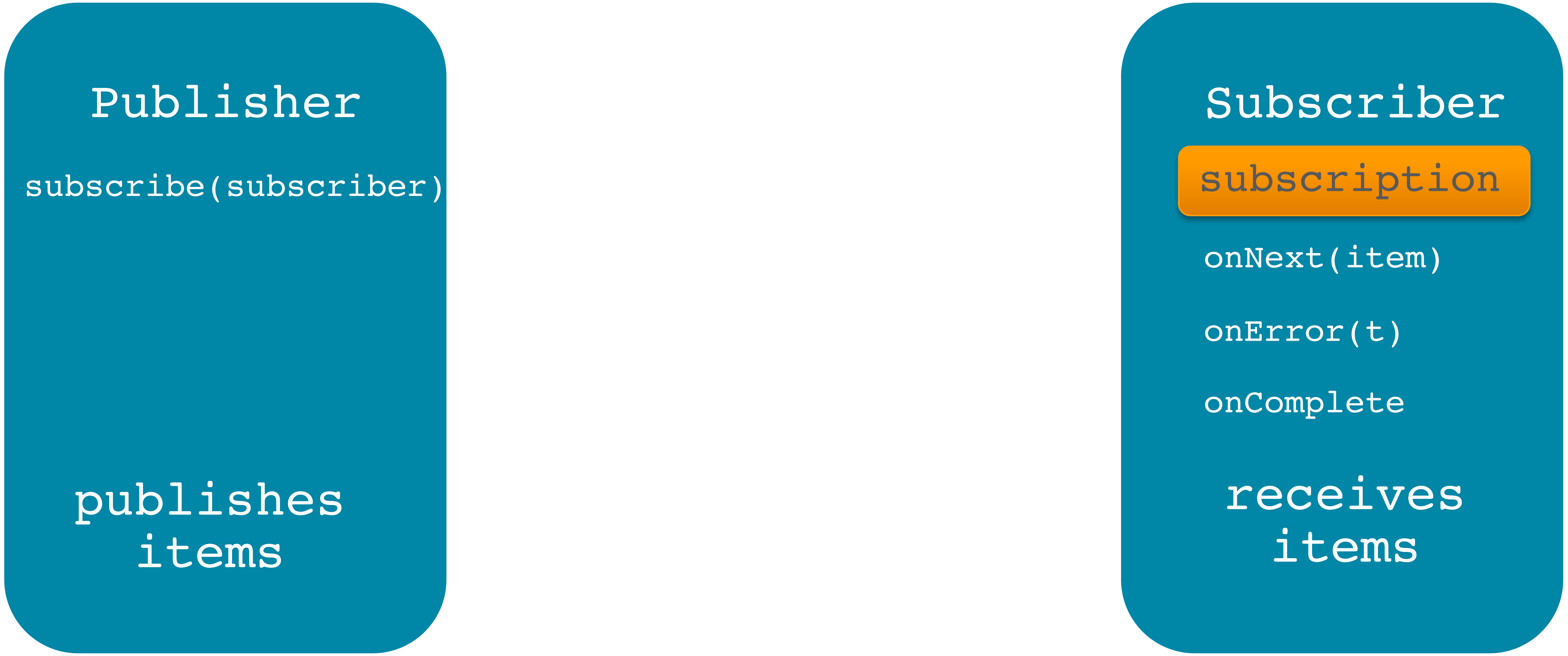
Reactive-streams (40th foot view)



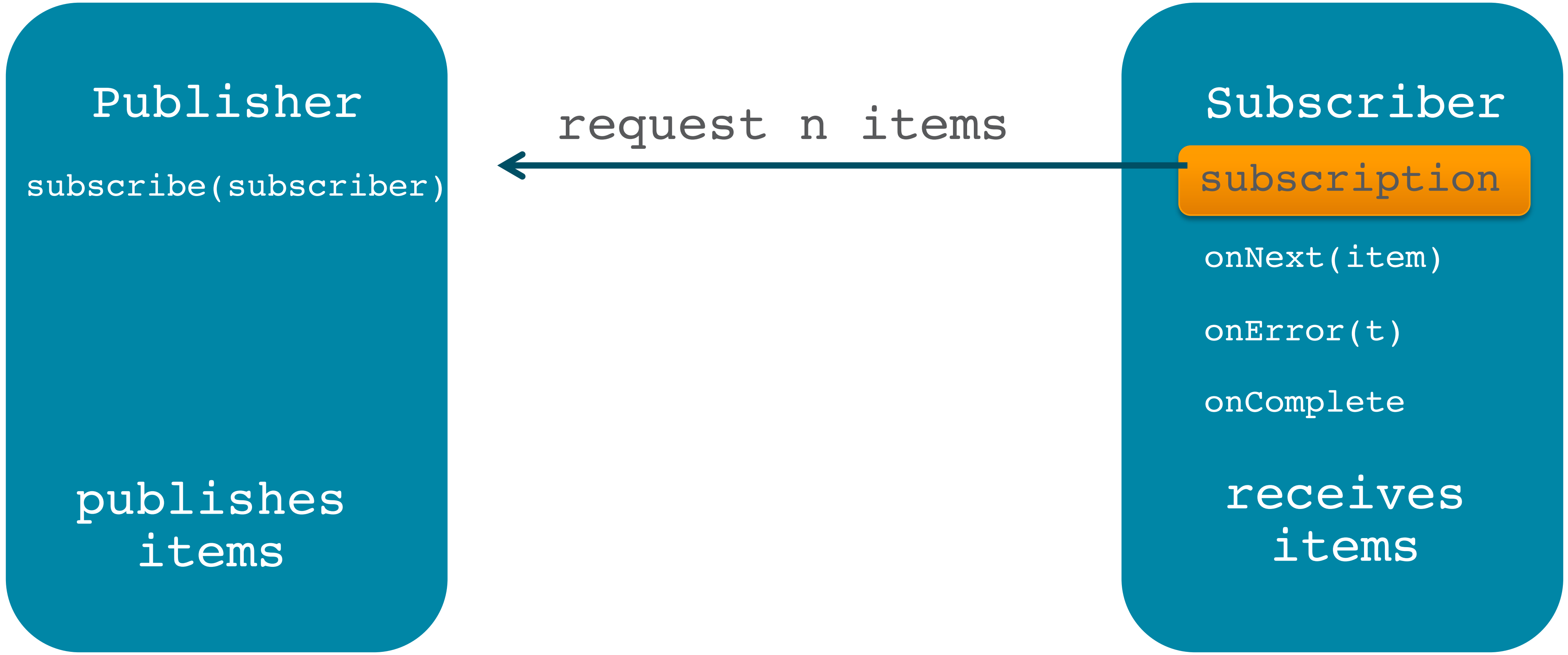
Reactive-streams (40th foot view)



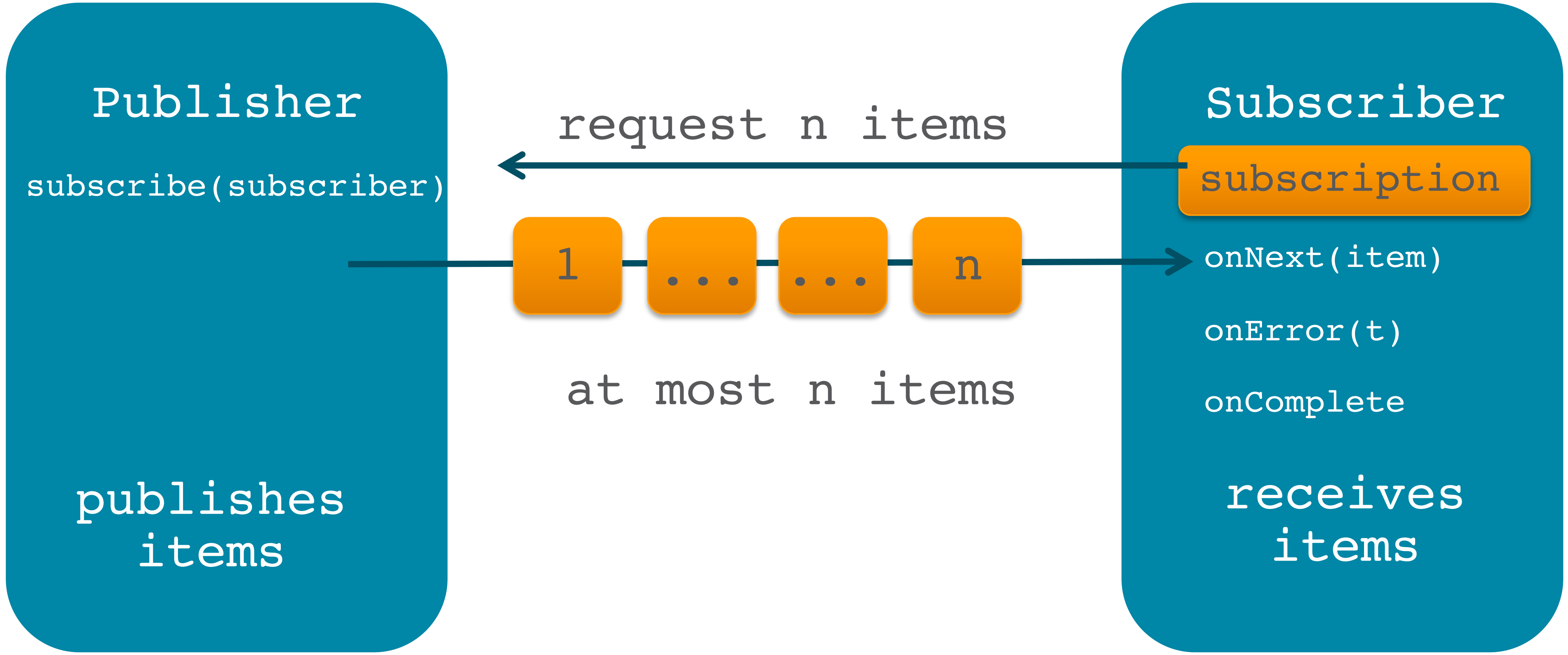
Reactive-streams (40th foot view)



Reactive-streams (40th foot view)



Reactive-streams (40th foot view)



Reactive-streams (40th foot view)

Publisher

`subscribe(subscriber)`

publishes items

Subscriber

`subscription`

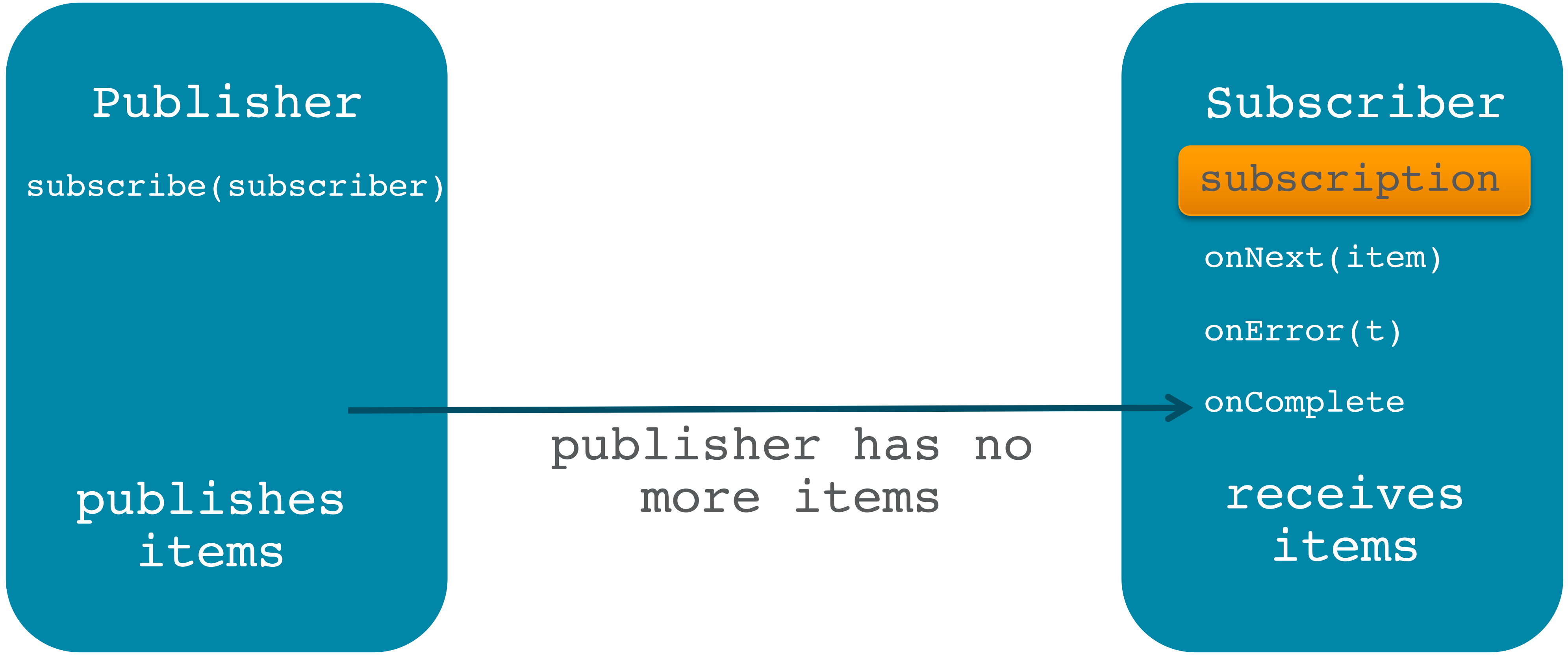
`onNext(item)`

`onError(t)`

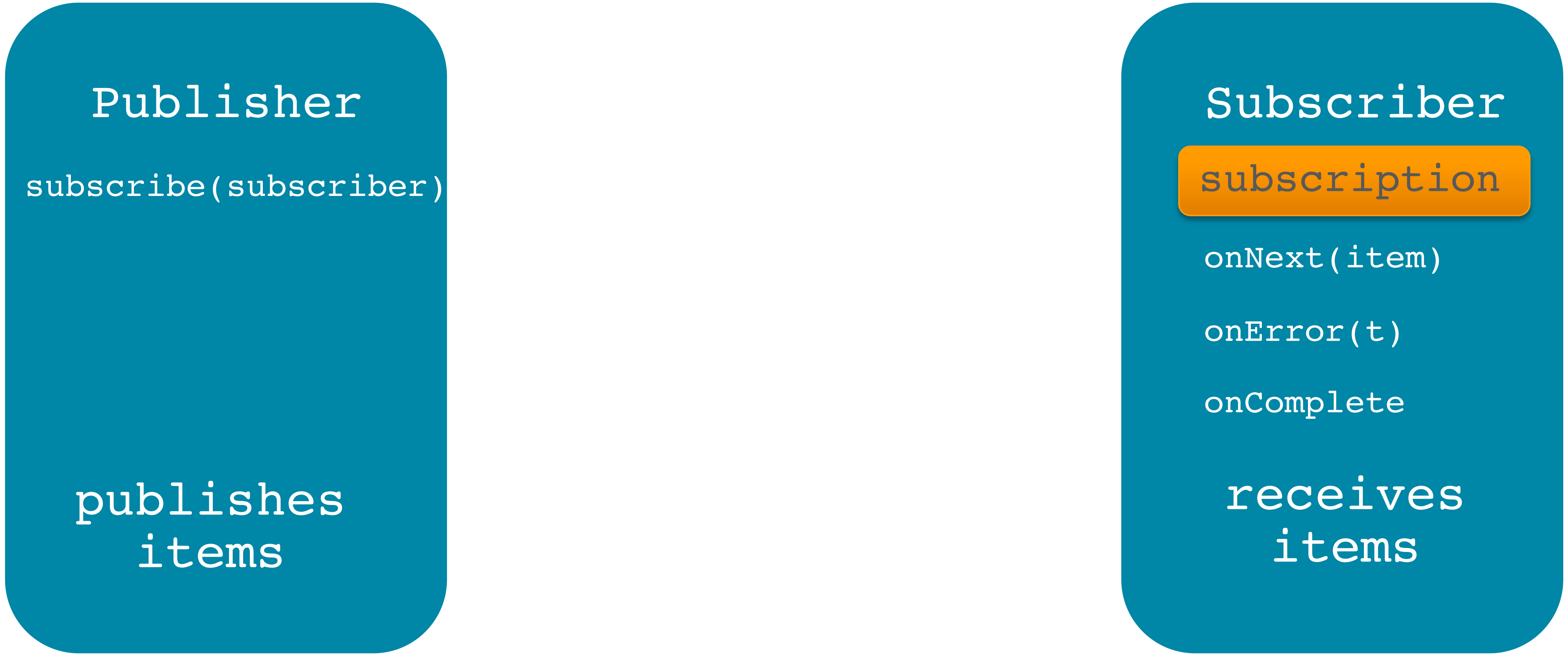
`onComplete`

receives items

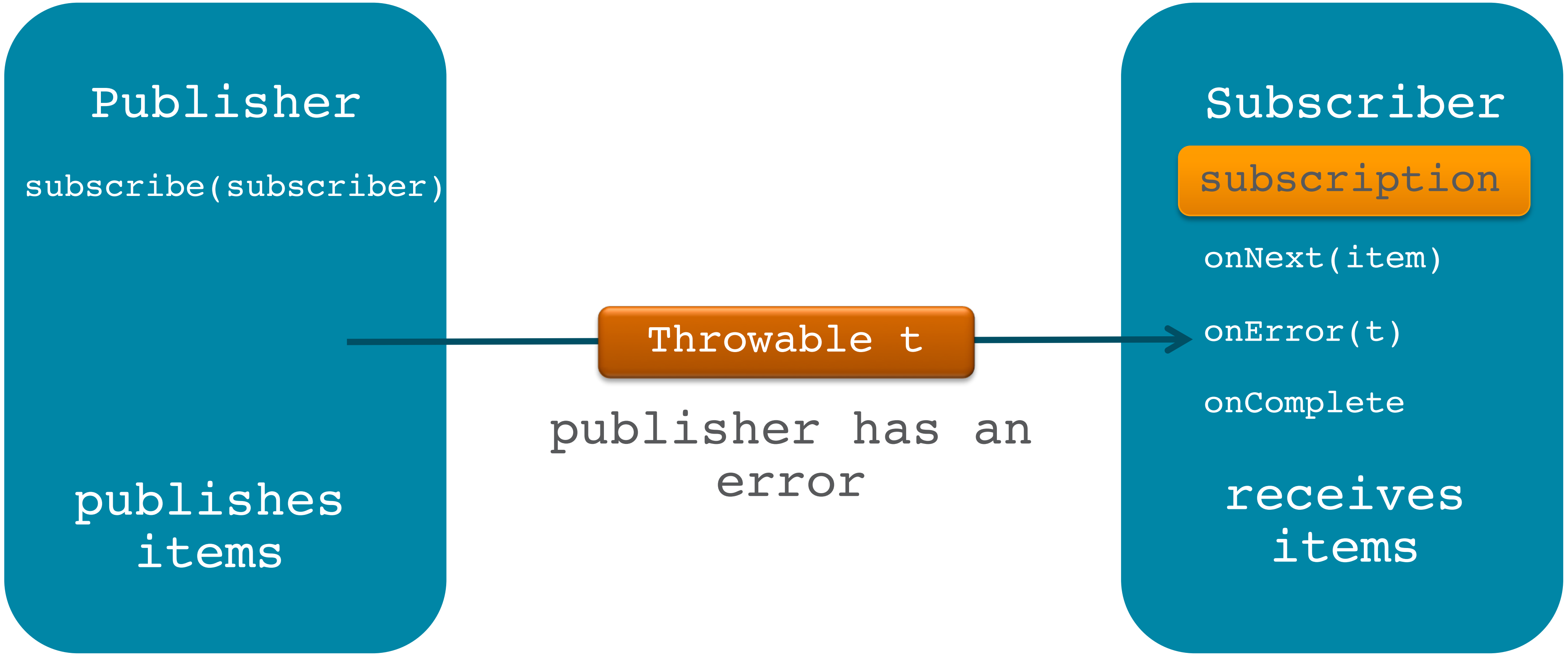
Reactive-streams (40th foot view)



Reactive-streams (40th foot view)



Reactive-streams (40th foot view)



Data as reactive-streams

- The request and response bodies are exposed as reactive streams (asynchronous streams of data with non-blocking back pressure.)
- The `HttpClient` is effectively a `Subscriber` of request body and a `Publisher` of response body bytes.
- The `BodyHandler` interface allows inspection of the response code and headers, before the actual response body is received, and is responsible for creating the response `BodySubscriber`.

HttpRequest.BodyPublisher

```
class HttpRequest {
    ...
    interface BodyPublisher
            extends Flow.Publisher<ByteBuffer> {
       long contentLength();
    }
}
```

- converts high-level Java objects into a flow of byte buffers suitable for sending as a request body

HttpRequest.BodyPublishers

Provides a number of convenience factory methods for creating request publishers for handling common body types such as files, Strings, and bytes.

- `BodyPubishers::ofByteArray(byte[])`
- `BodyPubishers::ofByteArrays(Iterable<byte[]>)`
- `BodyPubishers::ofFile(Path)`
- `BodyPubishers::ofString(String)`
- `BodyPubishers::ofInputStream(Supplier<InputStream>)`
- …

HttpResponse.BodyHandler

```
class HttpResponse<T> {
    ...
    interface ResponseInfo {
        int statusCode(); HttpHeaders headers(); ... }

    interface BodyHandler<T> {
        BodySubscriber<T> apply(ResponseInfo); }
}
```

- Allows inspection of the response code and headers, before the actual response body is received
- Responsible for creating the response BodySubscriber.

HttpResponse.BodySubscriber

```
class HttpResponse<T> {
    ...
    interface BodySubscriber<T>
        extends Flow.Subscriber<List<ByteBuffer>> {

        CompletionStage<T> getBody();
    }
}
```

- consumes response body bytes and converts them into a higher-level Java type.

HttpResponse.BodyHandlers

Provides a number of convenience factory methods for handling common response body types such as files, Strings, and bytes.

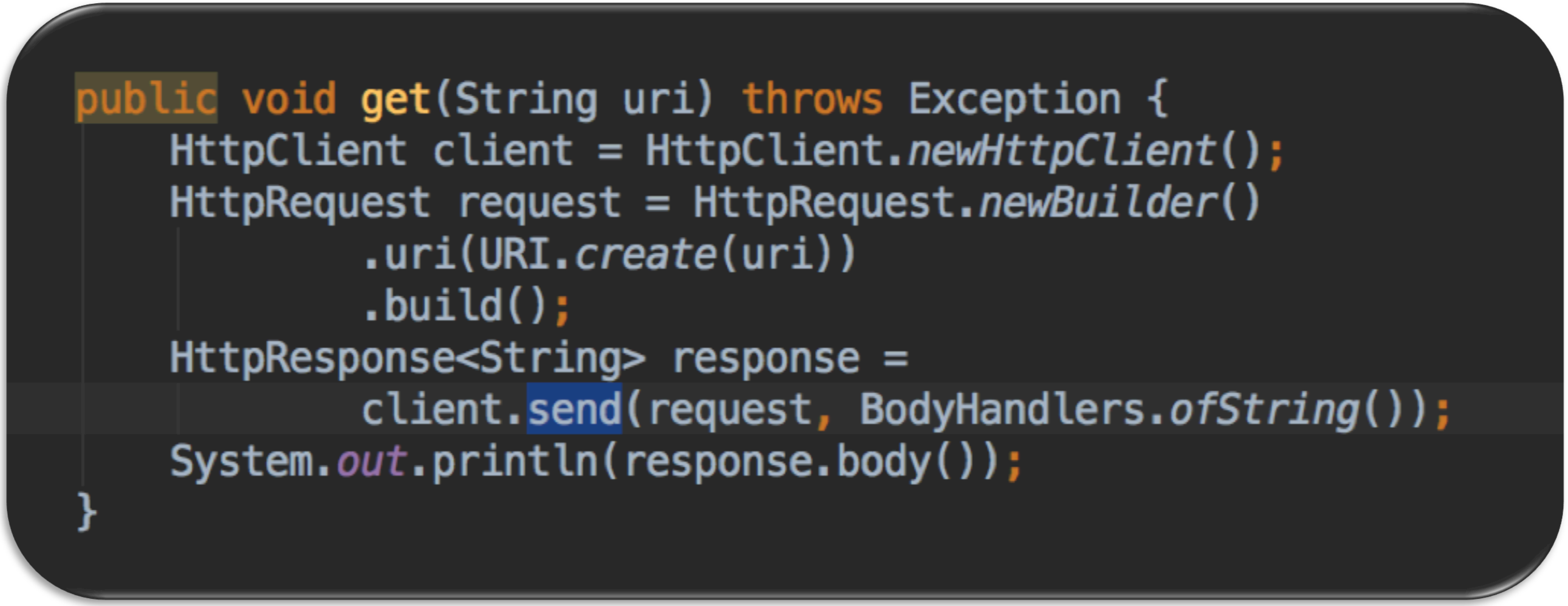
- `BodyHandlers::ofByteArray()`
- `BodyHandlers::ofFile()`
- `BodyHandlers::ofString()`
- `BodyHandlers::ofInputStream()`
- `BodyHandlers::replacing(U replacementValue)`
- `BodyHandlers::discarding()`
- `BodyHandlers::buffering(...)`
- …

Putting it all together

some examples…

Synchronous Get

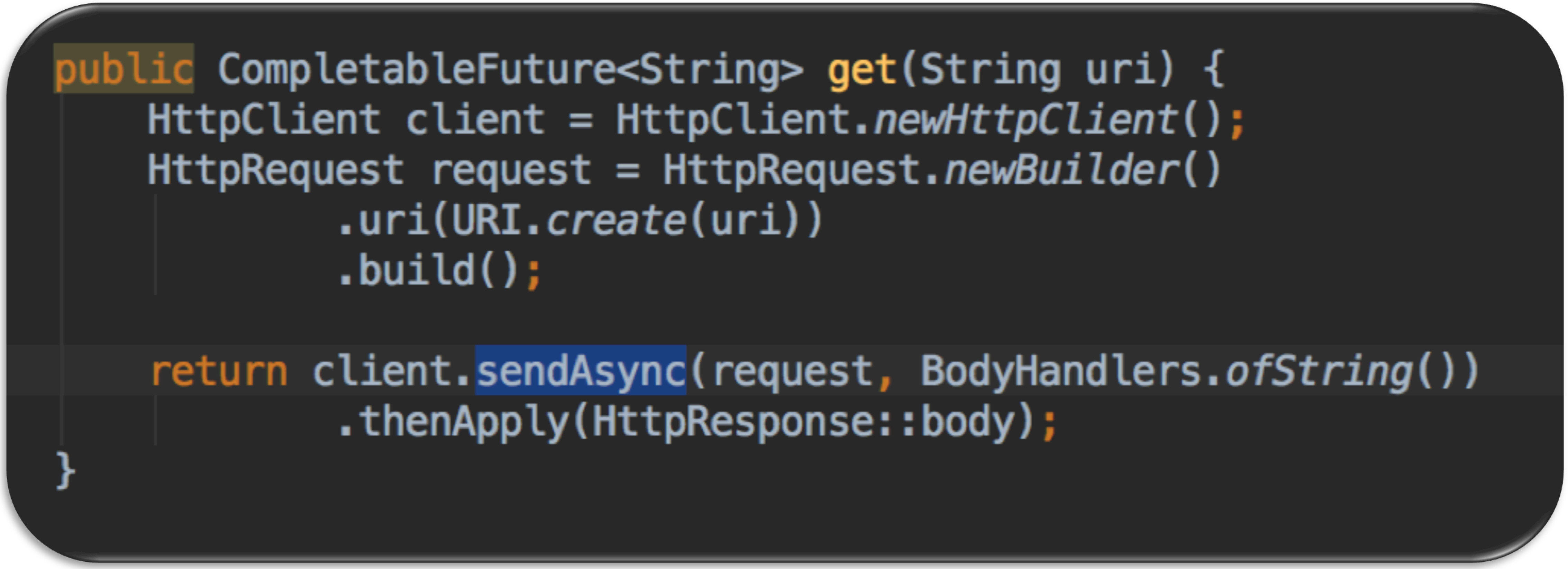
Response body as a String



```
public void get(String uri) throws Exception {
    HttpClient client = HttpClient.newHttpClient();
    HttpRequest request = HttpRequest.newBuilder()
            .uri(URI.create(uri))
            .build();
    HttpResponse<String> response =
            client.send(request, BodyHandlers.ofString());
    System.out.println(response.body());
}
```

Asynchronous Get

Response body as a String

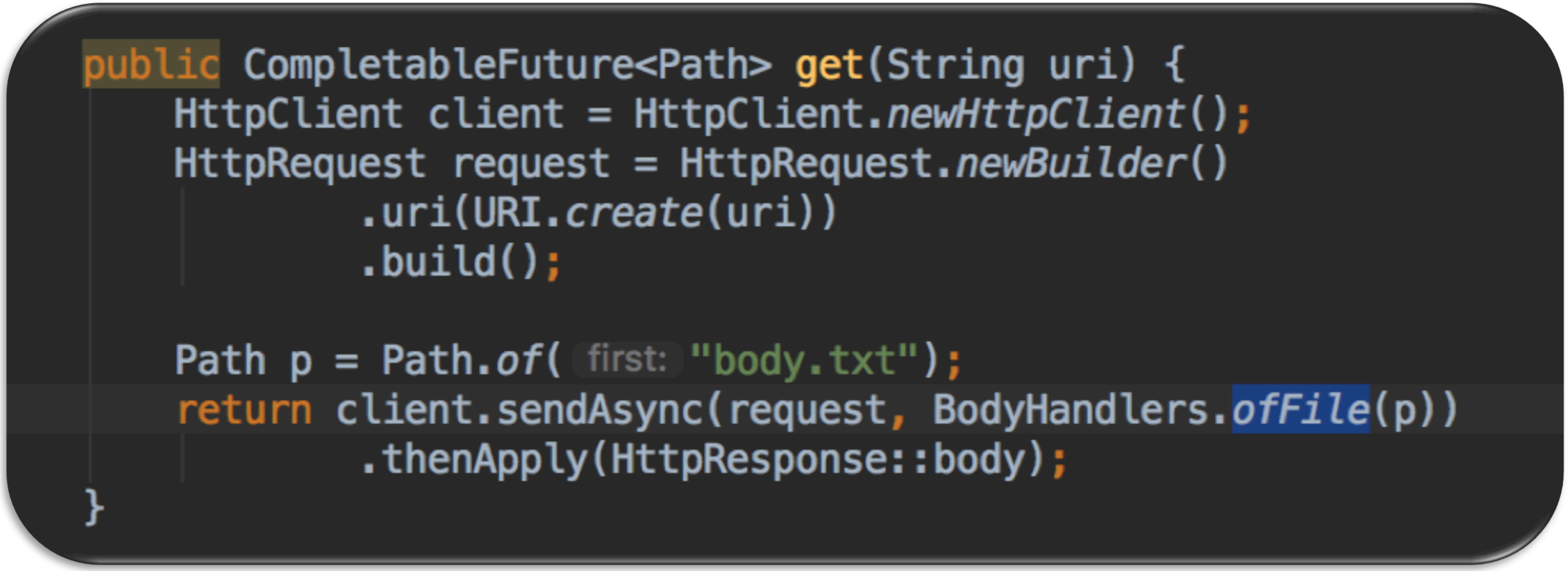


```java
public CompletableFuture<String> get(String uri) {
    HttpClient client = HttpClient.newHttpClient();
    HttpRequest request = HttpRequest.newBuilder()
            .uri(URI.create(uri))
            .build();

    return client.sendAsync(request, BodyHandlers.ofString())
            .thenApply(HttpResponse::body);
}
```

Asynchronous Get

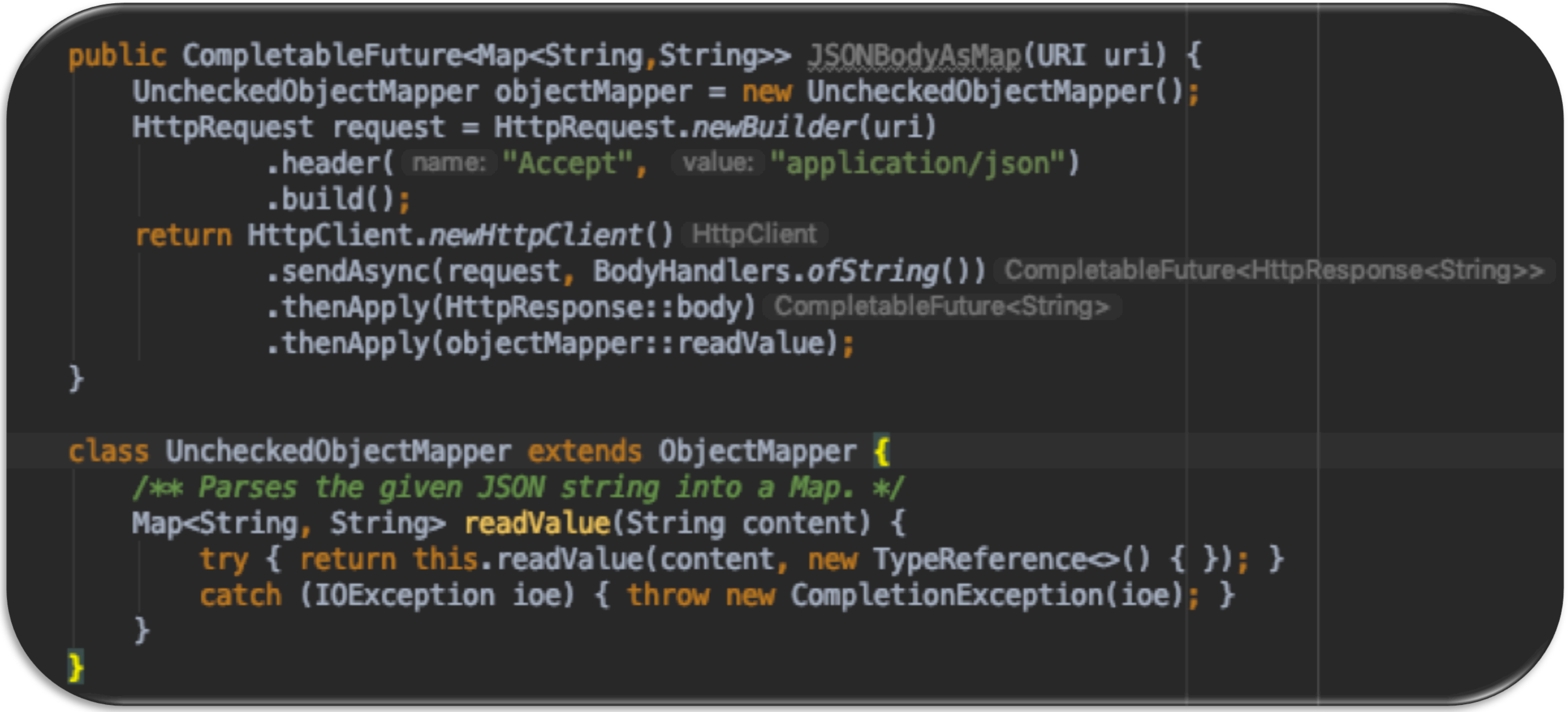
Response body as a File



```java
public CompletableFuture<Path> get(String uri) {
    HttpClient client = HttpClient.newHttpClient();
    HttpRequest request = HttpRequest.newBuilder()
            .uri(URI.create(uri))
            .build();

    Path p = Path.of( first: "body.txt");
    return client.sendAsync(request, BodyHandlers.ofFile(p))
            .thenApply(HttpResponse::body);
}
```

JSON Get

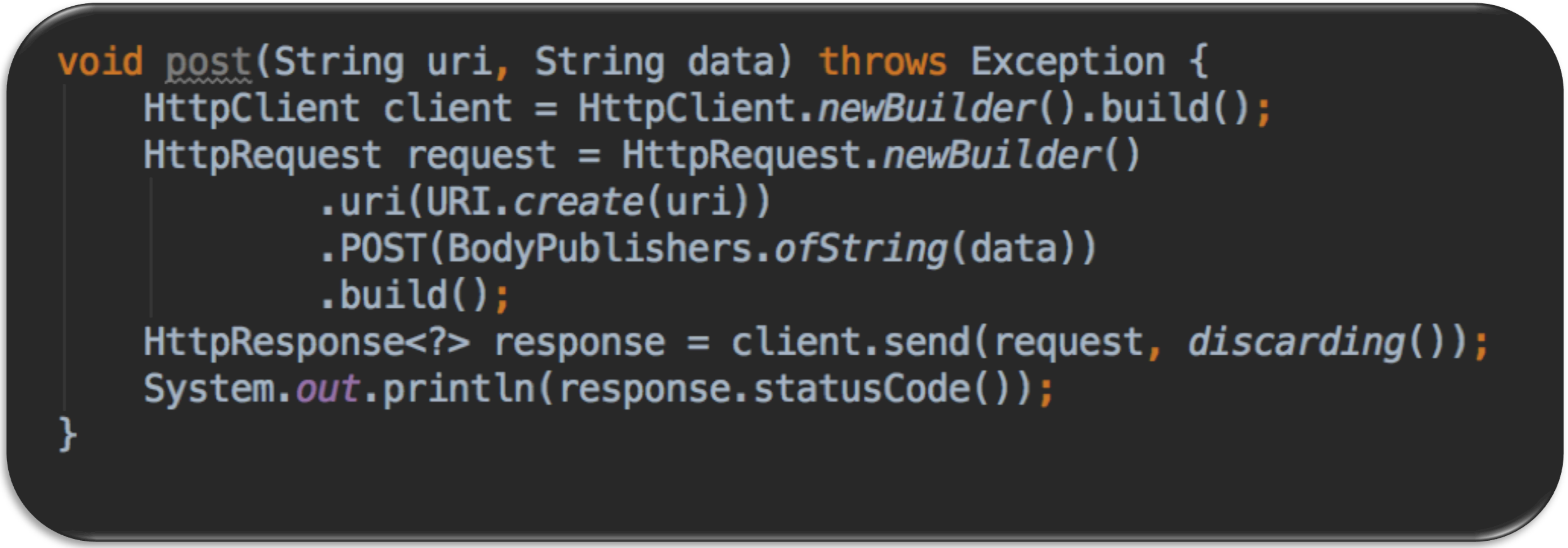


```
public CompletableFuture<Map<String,String>> JSONBodyAsMap(URI uri) {
    UncheckedObjectMapper objectMapper = new UncheckedObjectMapper();
    HttpRequest request = HttpRequest.newBuilder(uri)
            .header( name: "Accept",  value: "application/json")
            .build();
    return HttpClient.newHttpClient() HttpClient
            .sendAsync(request, BodyHandlers.ofString()) CompletableFuture<HttpResponse<String>>
            .thenApply(HttpResponse::body) CompletableFuture<String>
            .thenApply(objectMapper::readValue);
}

class UncheckedObjectMapper extends ObjectMapper {
    /** Parses the given JSON string into a Map. */
    Map<String, String> readValue(String content) {
        try { return this.readValue(content, new TypeReference<>() { }); }
        catch (IOException ioe) { throw new CompletionException(ioe); }
    }
}
```

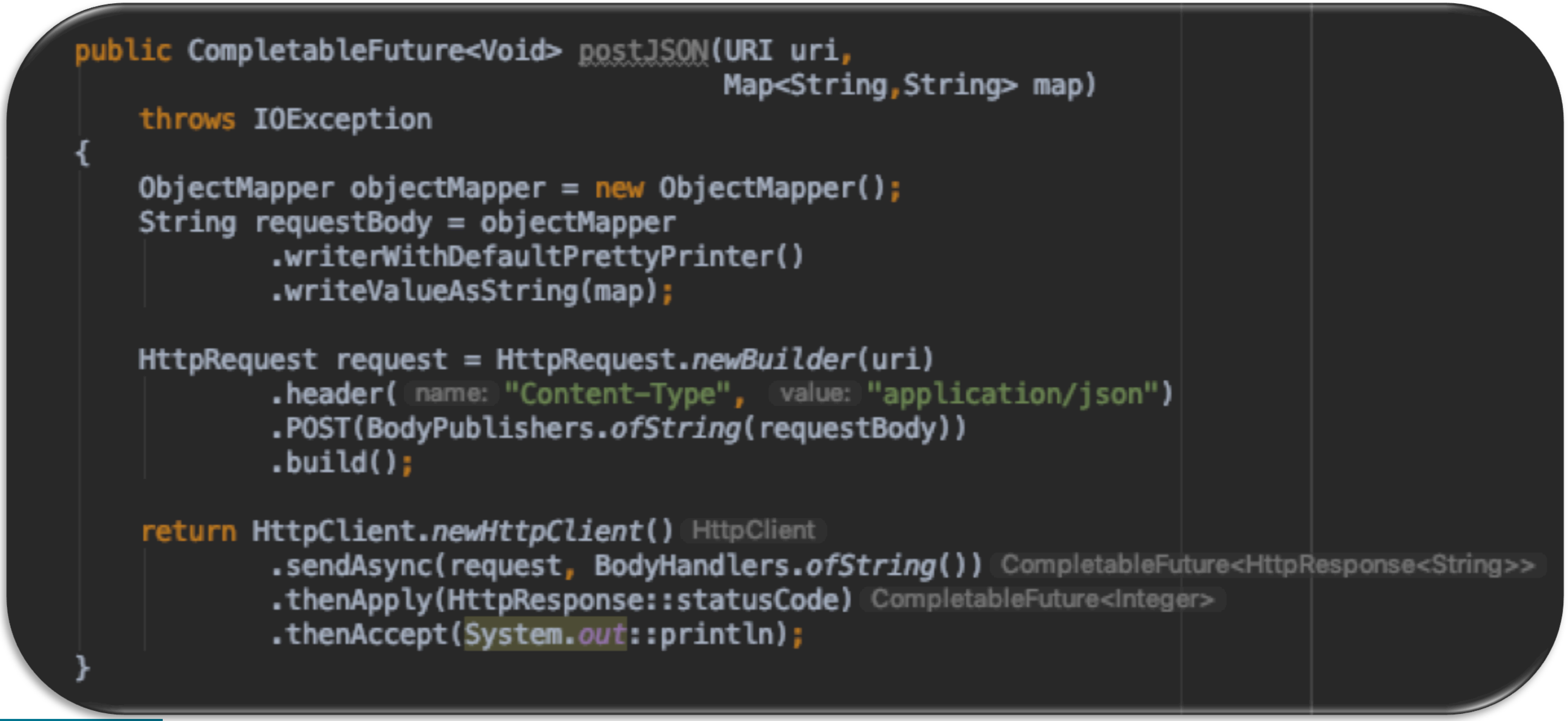
Post

A request body can be supplied by an `HttpRequest.BodyPublisher`.



```java
void post(String uri, String data) throws Exception {
    HttpClient client = HttpClient.newBuilder().build();
    HttpRequest request = HttpRequest.newBuilder()
            .uri(URI.create(uri))
            .POST(BodyPublishers.ofString(data))
            .build();
    HttpResponse<?> response = client.send(request, discarding());
    System.out.println(response.statusCode());
}
```

JSON Post



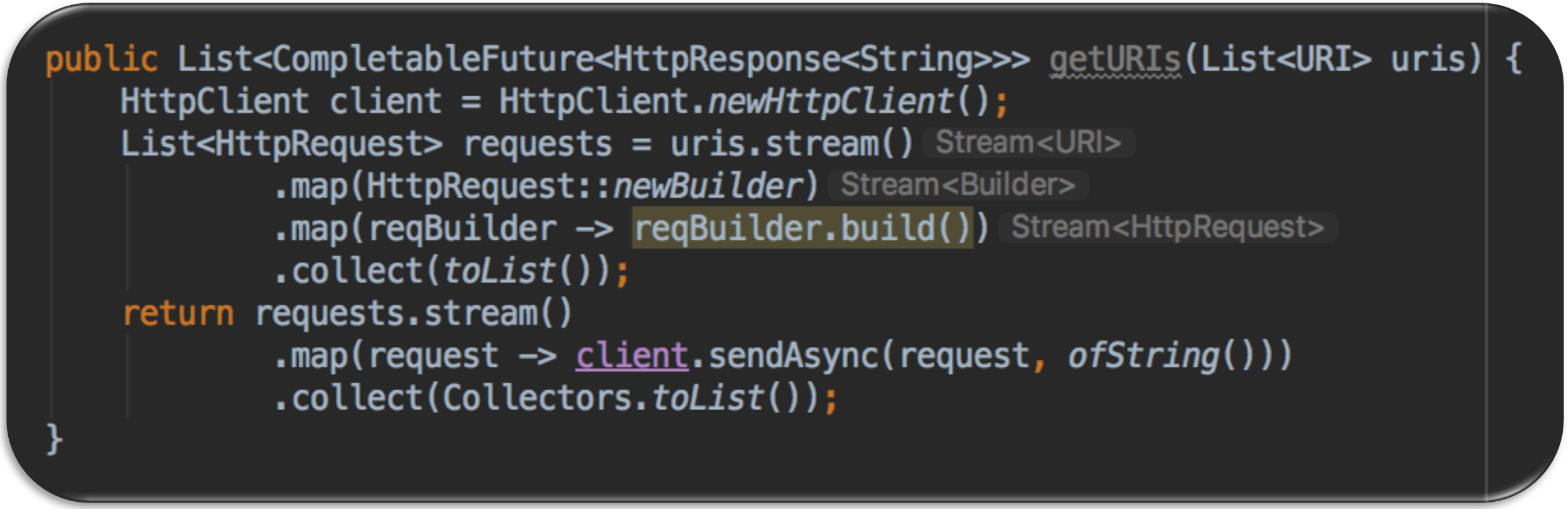
```
public CompletableFuture<Void> postJSON(URI uri,
                                        Map<String,String> map)
    throws IOException
{
    ObjectMapper objectMapper = new ObjectMapper();
    String requestBody = objectMapper
            .writerWithDefaultPrettyPrinter()
            .writeValueAsString(map);

    HttpRequest request = HttpRequest.newBuilder(uri)
            .header( name: "Content-Type",  value: "application/json")
            .POST(BodyPublishers.ofString(requestBody))
            .build();

    return HttpClient.newHttpClient() HttpClient
            .sendAsync(request, BodyHandlers.ofString()) CompletableFuture<HttpResponse<String>>
            .thenApply(HttpResponse::statusCode) CompletableFuture<Integer>
            .thenAccept(System.out::println);
}
```

Concurrent Requests

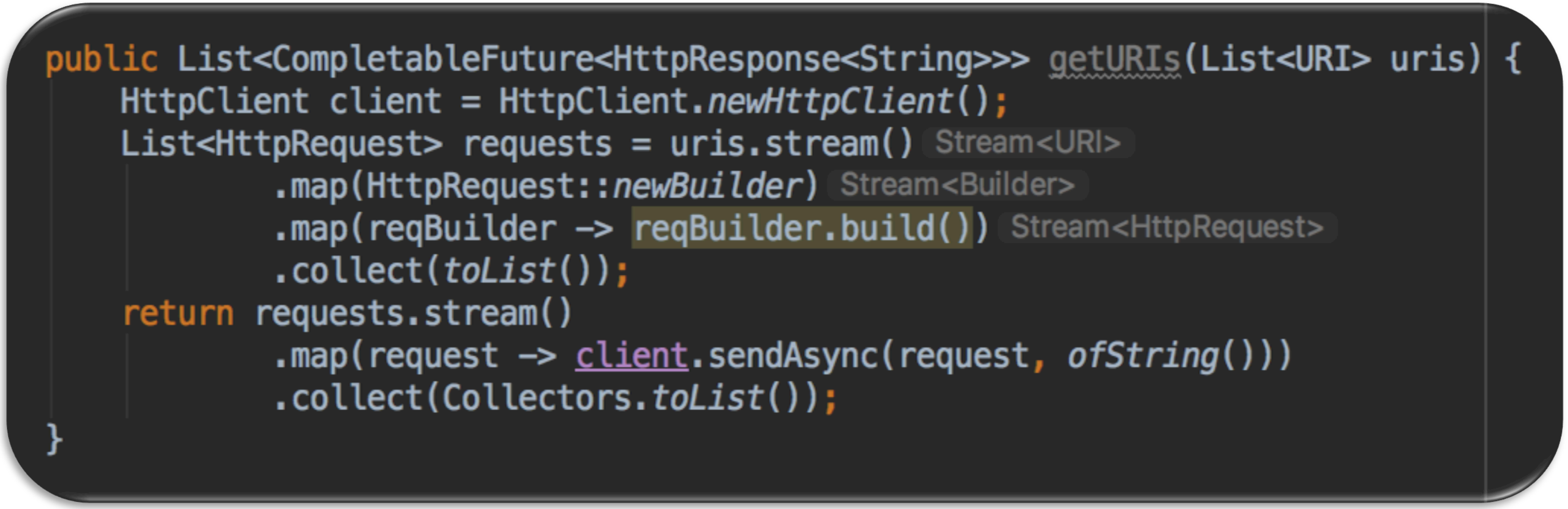
It's easy to combine Java Streams and the CompletableFuture API to issue a number of requests and await their responses.



```java
public List<CompletableFuture<HttpResponse<String>>> getURIs(List<URI> uris) {
    HttpClient client = HttpClient.newHttpClient();
    List<HttpRequest> requests = uris.stream() Stream<URI>
            .map(HttpRequest::newBuilder) Stream<Builder>
            .map(reqBuilder -> reqBuilder.build()) Stream<HttpRequest>
            .collect(toList());
    return requests.stream()
            .map(request -> client.sendAsync(request, ofString()))
            .collect(Collectors.toList());
}
```

Concurrent Requests

Sends a GET request for each of the URIs in the list and stores all the responses as Strings.



```java
public List<CompletableFuture<HttpResponse<String>>> getURIs(List<URI> uris) {
    HttpClient client = HttpClient.newHttpClient();
    List<HttpRequest> requests = uris.stream() Stream<URI>
            .map(HttpRequest::newBuilder) Stream<Builder>
            .map(reqBuilder -> reqBuilder.build()) Stream<HttpRequest>
            .collect(toList());
    return requests.stream()
            .map(request -> client.sendAsync(request, ofString()))
            .collect(Collectors.toList());
}
```

Handling response data

1. **How to write a custom response BodySubscriber.**

2. **Interoperability with other reactive streams implementations.**

Writing a custom response BodySubscriber

demo

https://www.youtube.com/watch?v=qiaC0QMLz5Y

Interoperability with reactive streams

- `HttpResponse`
  - `BodySubscribers::fromSubscriber(Flow.Subscriber)`
  - `BodySubscribers::fromSubscriber(Flow.Subscriber,`
    `Function finisher)`
  - `BodySubscribers::fromLineSubscriber(Flow.Subscriber)`
  - `BodySubscribers::fromLineSubscriber(Flow.Subscriber`
    `Function finisher)`
  - `BodySubscribers::ofPublisher()`

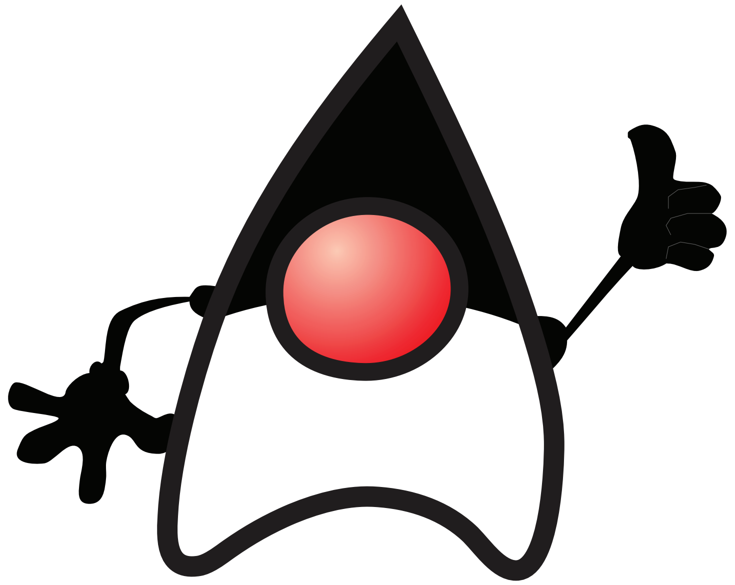
- `HttpRequest`
  - `BodyPublishers::fromPublisher(Flow.Publisher)`
  - `BodyPublishers::fromPublisher(Flow.Publisher, long cl)`

Summary

The HTTP Client

- Replacement for the URLConnection API
- Incubated in JDK 9; Refreshed in JDK 10; Standardized in Java 11
- Code using the incubated version will need to be updated a little for Java 11 (`jdk.incubator.http -> java.net.http`)
- Small compact API
- Uses modern Java language and API features,
  - Generics, lambdas, CompletableFuture, Reactive-streams
- Learned about how reactive streams are used by the HTTP Client
- Interoperability with existing reactive streams implementations

- Download: jdk.java.net or www.oracle.com/javadownload
- Join the OpenJDK: openjdk.java.net
- Follow us on Twitter: @OpenJDK, #Java,
  or at my twitter handle @chegar999



References

JEP 1 - JDK Enhancement-Proposal - https://openjdk.java.net/jeps/1
JEP 11 - Incubator Modules - https://openjdk.java.net/jeps/11
JEP 110 – HTTP/2 Client (Incubator) - https://openjdk.java.net/jeps/110
JEP 321 – HTTP Client (Standard) - https://openjdk.java.net/jeps/321
Java HTTP Client - https://openjdk.java.net/groups/net/httpclient/

Java™

ORACLE®