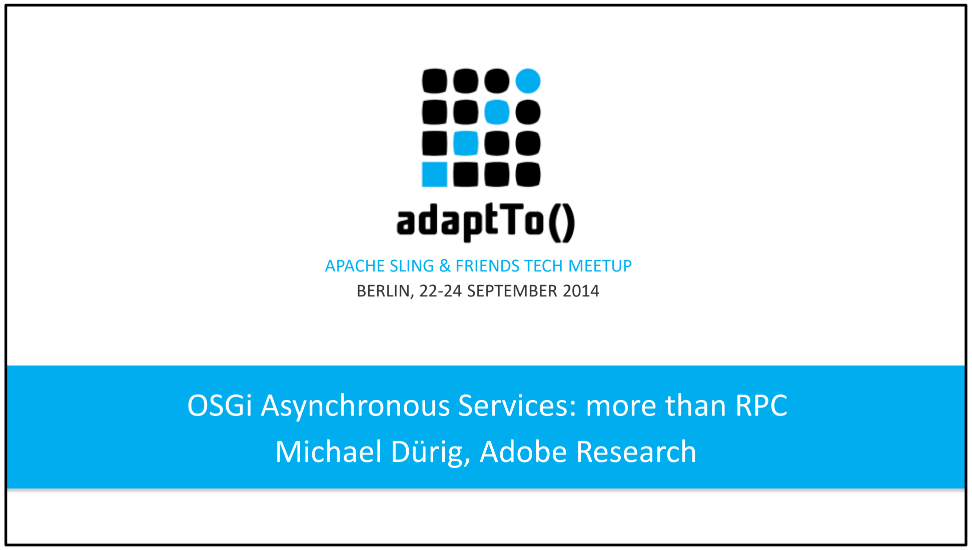


adaptTo()

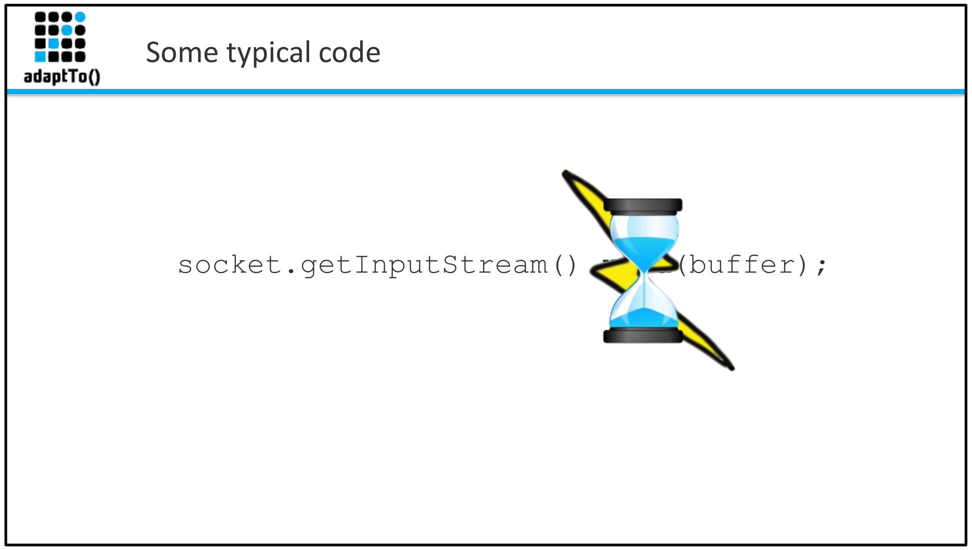
APACHE SLING & FRIENDS TECH MEETUP

BERLIN, 22-24 SEPTEMBER 2014

OSGi Asynchronous Services: more than RPC

Michael Dürig, Adobe Research

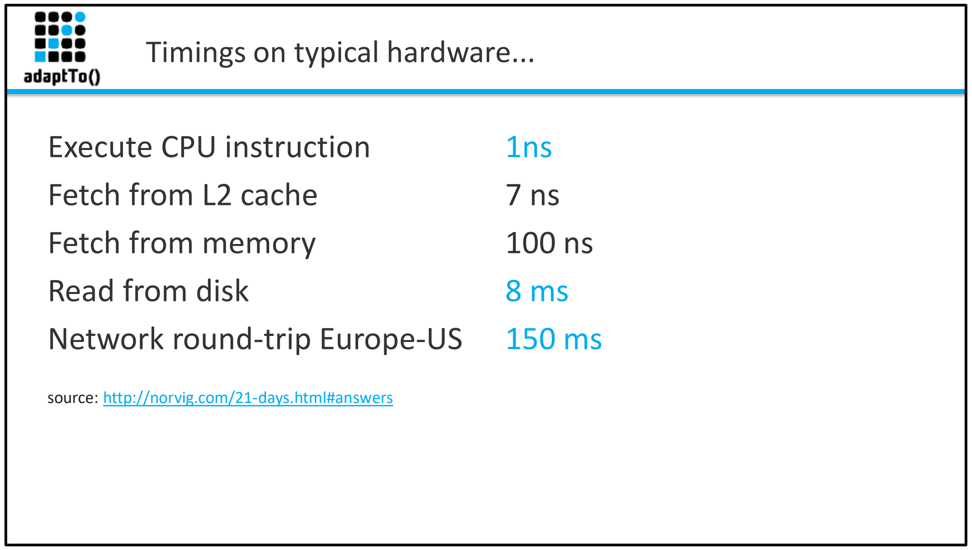
- RFC 206: https://github.com/osgi/design/tree/master/rfcs/rfc0206
- Work in progress. Emphasis on general concepts and Promises (2nd part)



Some typical code

```
socket.getInputStream().read(buffer);
```

- Side effects:
  - Might throw an exception
  - Blocks waiting for more data
- Blocking is not evident from the API signature: easy to forget about it
- Handling side effects is awkward:
  - Exception handling is difficult and blurs the main code path.
  - Not blocking is next to impossible in with this style.

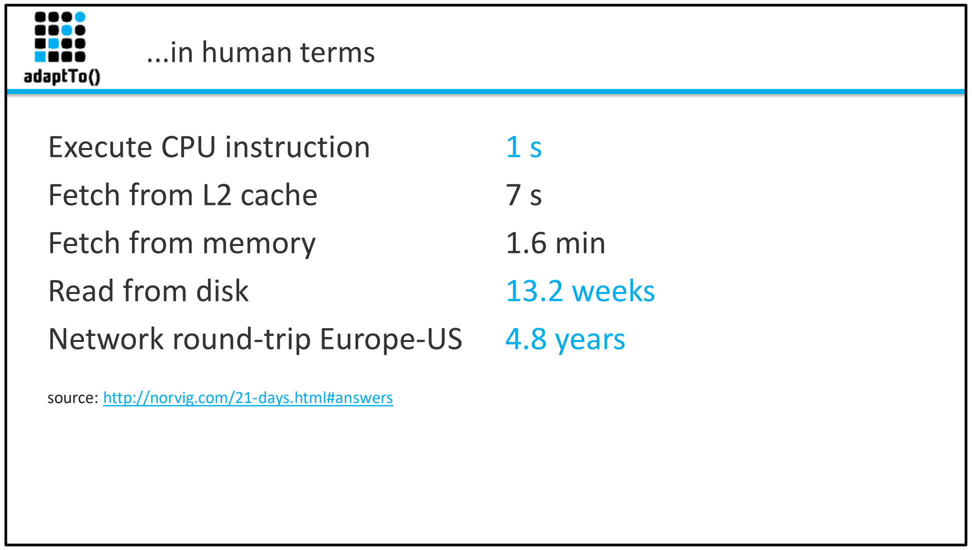


Timings on typical hardware...

Execute CPU instruction	1ns
Fetch from L2 cache	7 ns
Fetch from memory	100 ns
Read from disk	8 ms
Network round-trip Europe-US	150 ms

source: http://norvig.com/21-days.html#answers

Blocking **is** an issue!

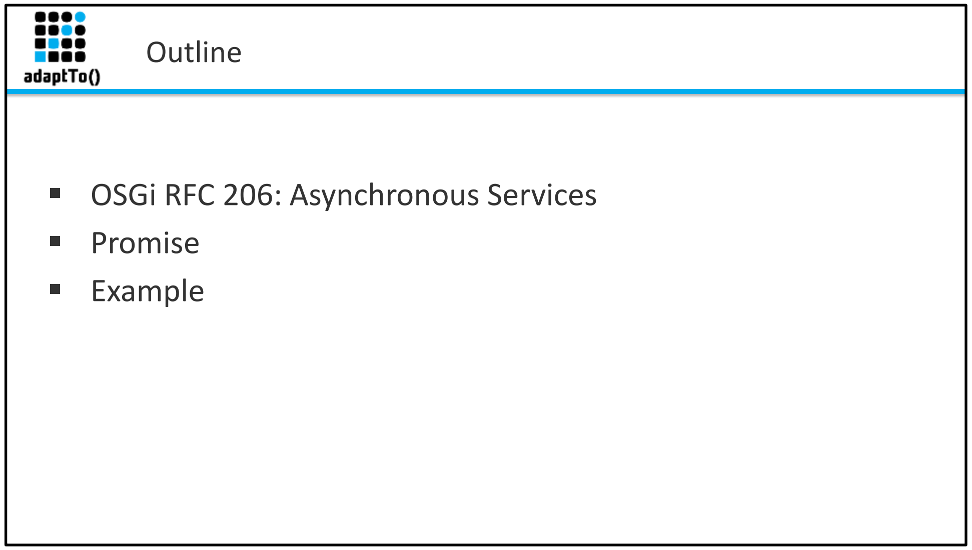


...in human terms

Execute CPU instruction	1 s
Fetch from L2 cache	7 s
Fetch from memory	1.6 min
Read from disk	13.2 weeks
Network round-trip Europe-US	4.8 years

source: http://norvig.com/21-days.html#answers

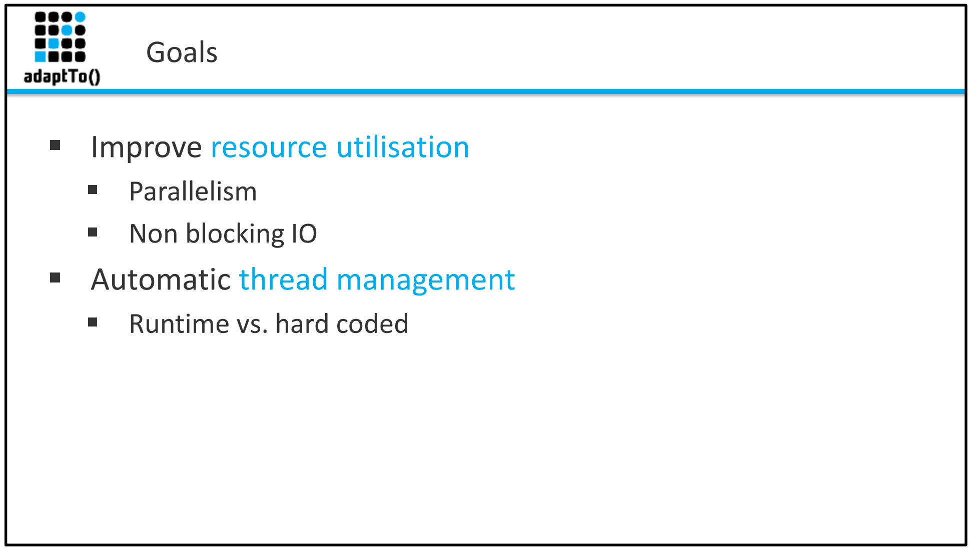
Scaling up to human terms results in almost 5 years spent waiting. Think of what could be done during that time!



Outline

- OSGi RFC 206: Asynchronous Services
- Promise
- Example

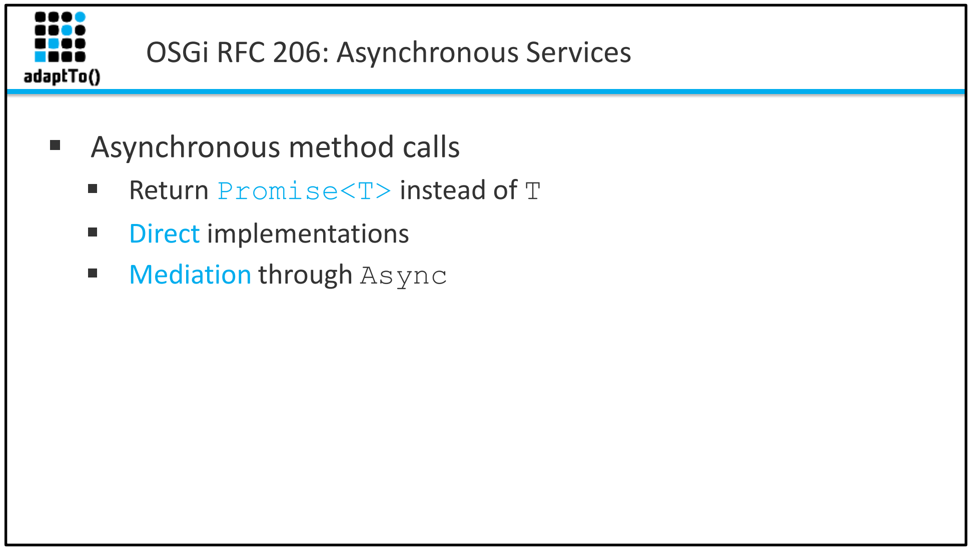
RFC 206 Asynchronous services provide ways to call methods asynchronously. By introducing the concept of a Promise they addresses the problems with exception handling and blocking.



Goals

- Improve resource utilisation
  - Parallelism
  - Non blocking IO
- Automatic thread management
  - Runtime vs. hard coded

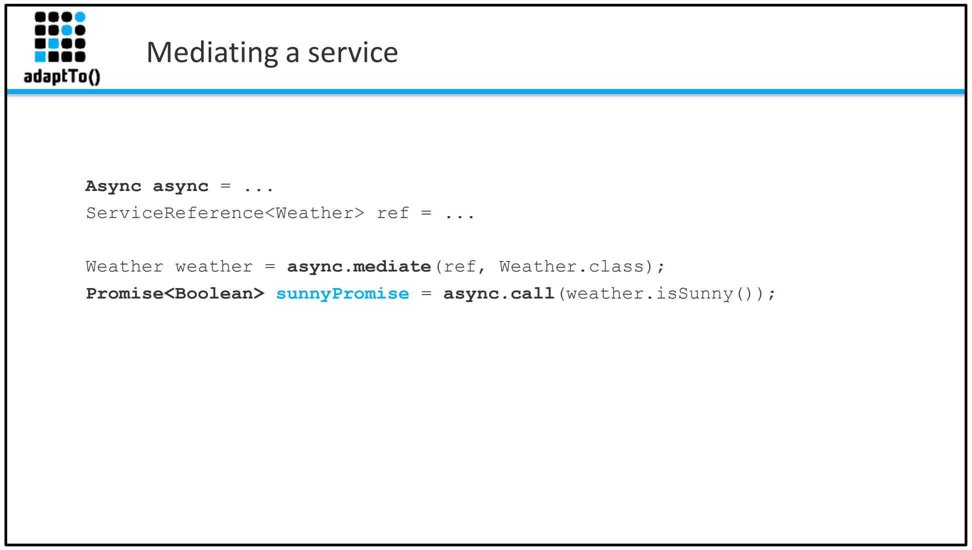
- Maximise application throughput and resource utilisation by submitting tasks for execution and letting the environment decide on how, when and where to execute them.
- Optimise resource utilisation by letting the environment assign threads to the units of parallelisation instead of managing threads manually in the application. The environment is in much better control of the available resources. This is similar to the shift from manual memory management to automatic memory management that took place in the 90ies.



OSGi RFC 206: Asynchronous Services

- Asynchronous method calls
  - Return `Promise<T>` instead of `T`
  - Direct implementations
  - Mediation through `Async`

- Asynchronous calls are not handles “transparently” as common in RPC.
- The side effects concerning remote calls are reflected in the return type: `Promise<T>` instead of `T`
- While any OSGi service can be converted to an asynchronous service via a mediation service, direct implementations are encouraged. The only requirement for such a service is to return a Promise.
- The Promise API can be used independently of the rest of the OSGi framework.

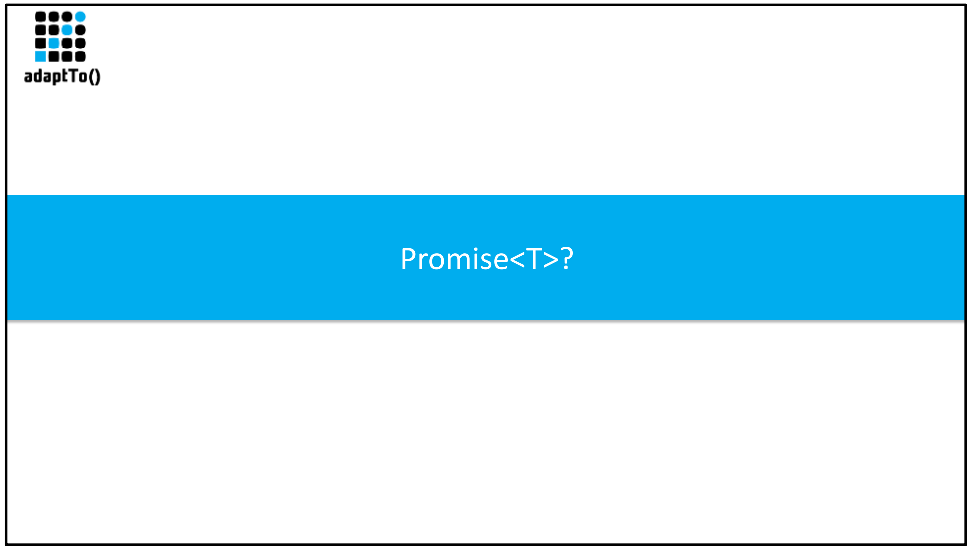


adaptTo()

Mediating a service

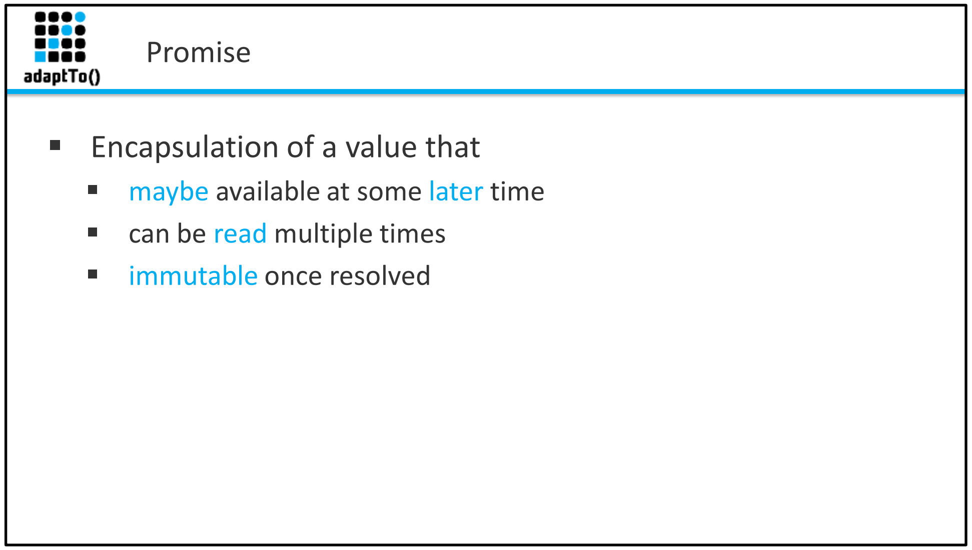
```
Async async = ...
ServiceReference<Weather> ref = ...

Weather weather = async.mediate(ref, Weather.class);
Promise<Boolean> sunnyPromise = async.call(weather.isSunny());
```

- `T Async#mediate(ServiceReference<T>, Class<T>)` returns an asynchronous service for `T`. For any method on `T` returning some type `R`, the asynchronous variant will return `Promise<R>`.
- `Promise<R> call(R r)` calls an asynchronous service returning a `Promise<R>` as a handle for the result.



Promise<T>?

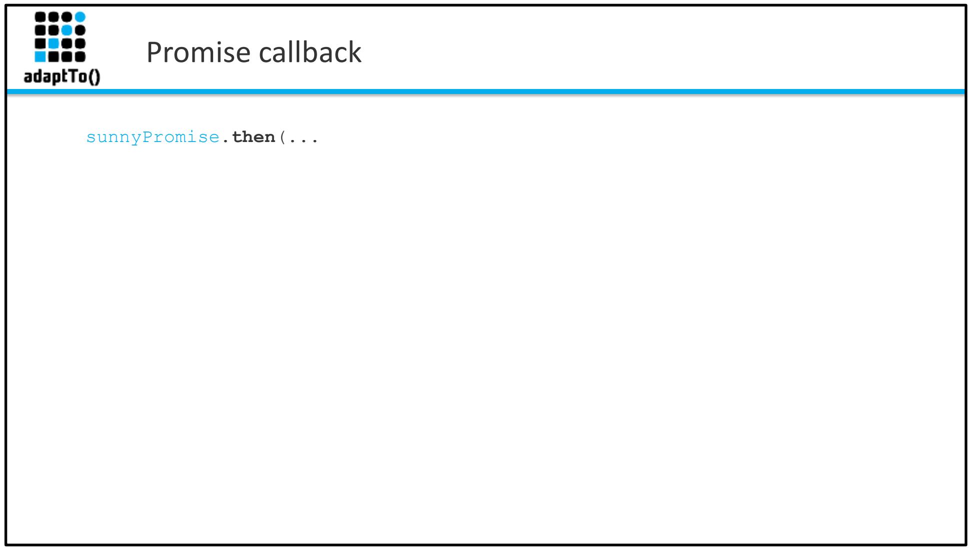
Unfortunately there is some confusion regarding the naming. The term Promise was inherited from Javascript and refers to what in the Java is usually called a Future. See the slide about Deferred in the Appendix for more confusion.



Promise

- Encapsulation of a value that
  - maybe available at some later time
  - can be read multiple times
  - immutable once resolved

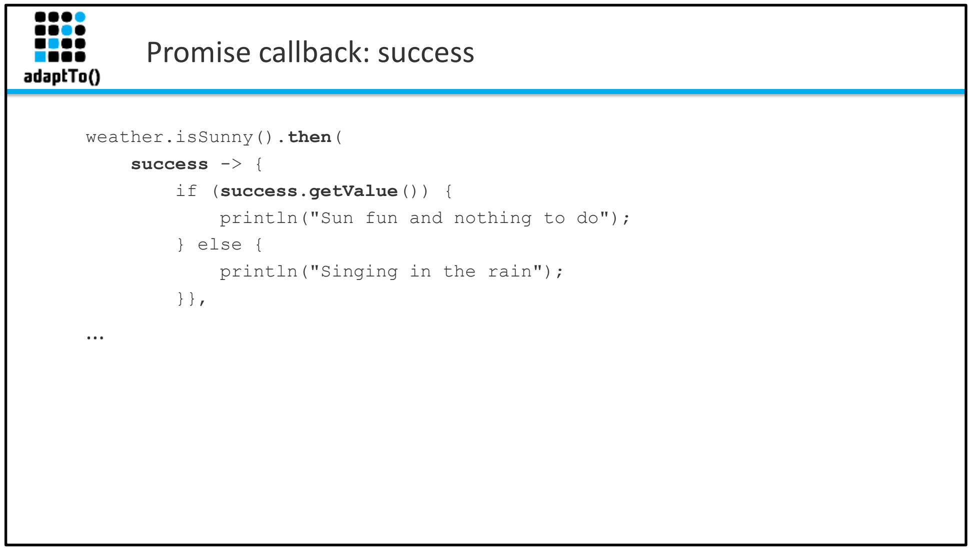
- Promises let you register completion call backs, which is the low level mechanism behind a much richer combinator API (map, flatMap, filter).
- A Promise is similar to a Java Future but done right: promises are composable through their various combinator methods and don’t require blocking for the result.
- A Promise effectively establishes a happened before relation: the completion of a Promise happens before the invocation of any callbacks.



Promise callback

```
sunnyPromise.then(...
```

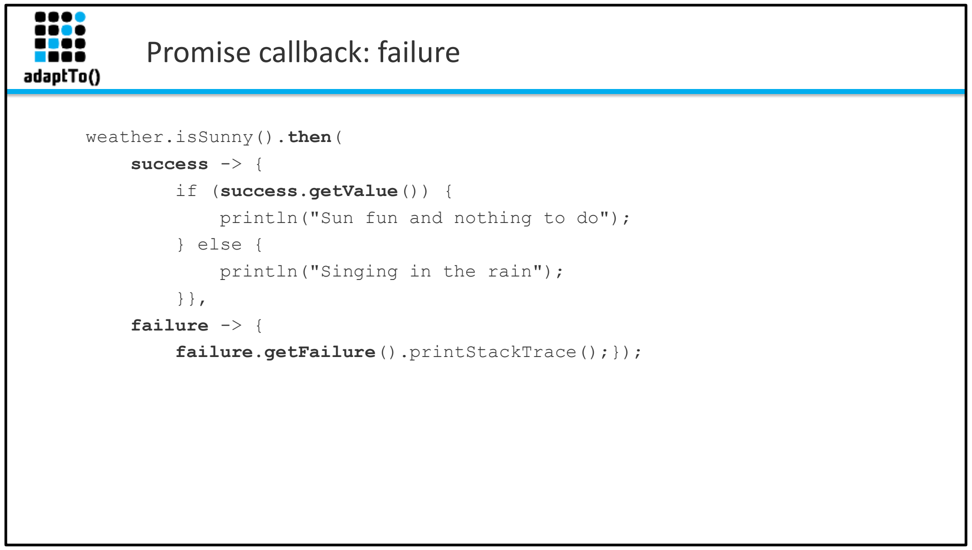
A call back consists of a success and a failure part. Depending on whether the Promise **resolves successfully** or **resolves with a failure** either of both callbacks will be called.



Promise callback: success

```
weather.isSunny().then(
    success -> {
        if (success.getValue()) {
            println("Sun fun and nothing to do");
        } else {
            println("Singing in the rain");
        }},
...
```

The success callback receives the successfully resolved promise. In this example it just retrieves the value and prints an according message. In general the success callback can return a chained promise, which will be used to resolve the promise returned by the then() call. That is, the promise returned by then() will resolve successfully if this promise resolve successfully and the promise returned by the success callback resolves successfully.

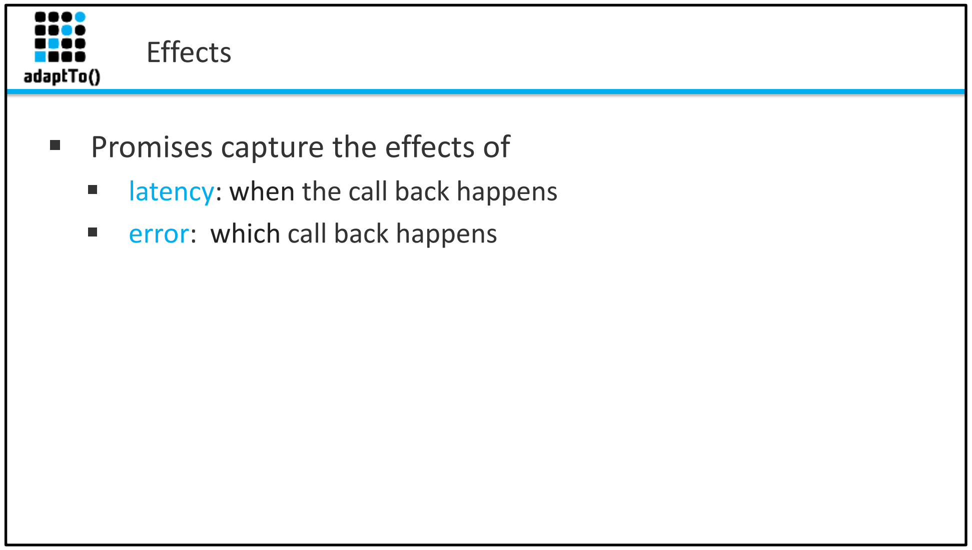


adaptTo()

Promise callback: failure

```
weather.isSunny().then(
    success -> {
        if (success.getValue()) {
            println("Sun fun and nothing to do");
        } else {
            println("Singing in the rain");
        }},
    failure -> {
        failure.getFailure().printStackTrace();});
```

The failure callback receives the failed promise. In this example it just prints the stack trace of the associated exception.



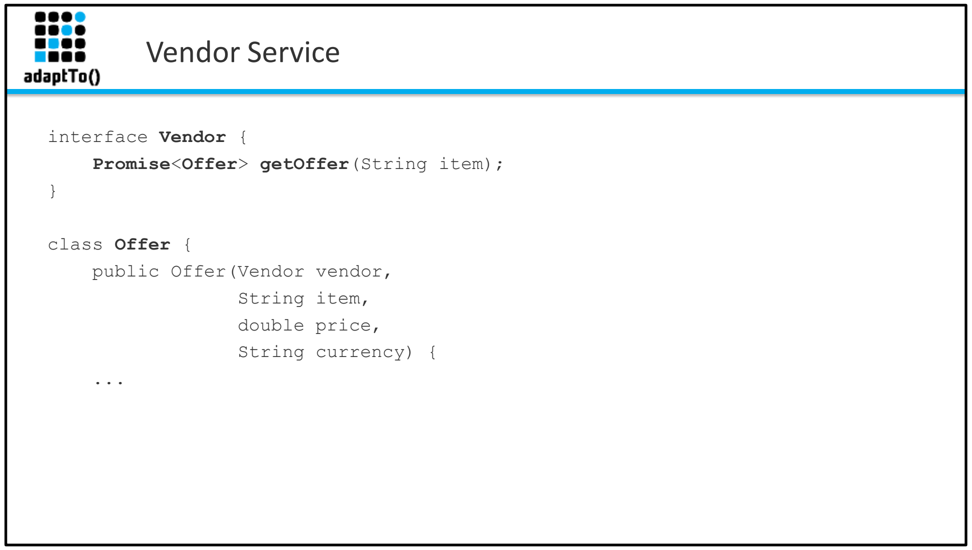
Effects

- Promises capture the effects of
  - latency: when the call back happens
  - error: which call back happens

Promises make the effects of e.g. remote calls explicit in the API signature. This is in contrast to classical RPC style, which tries to make such effects transparent and which usually does not work so well.

adaptTo()

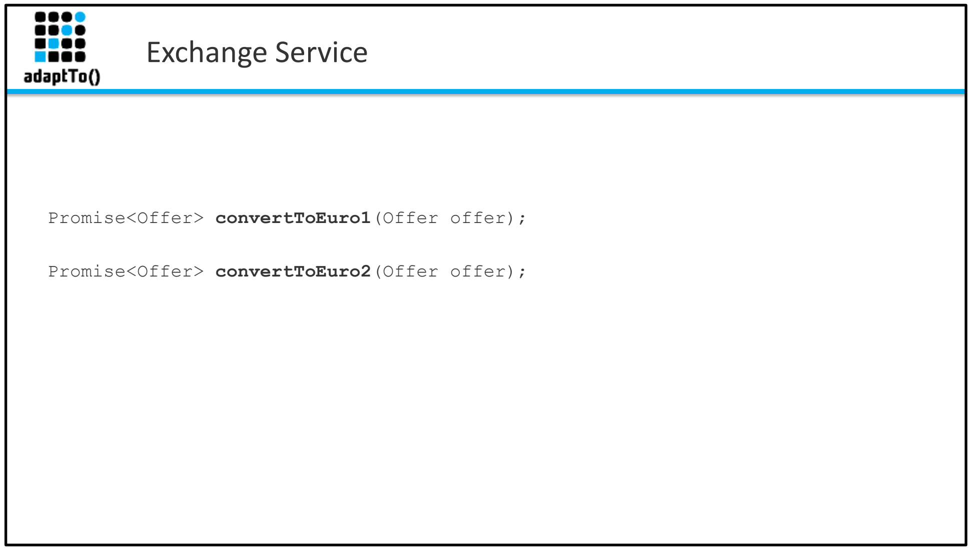
Promise<T>!



Vendor Service

```
interface Vendor {
    Promise<Offer> getOffer(String item);
}

class Offer {
    public Offer(Vendor vendor,
                 String item,
                 double price,
                 String currency) {
    ...
```

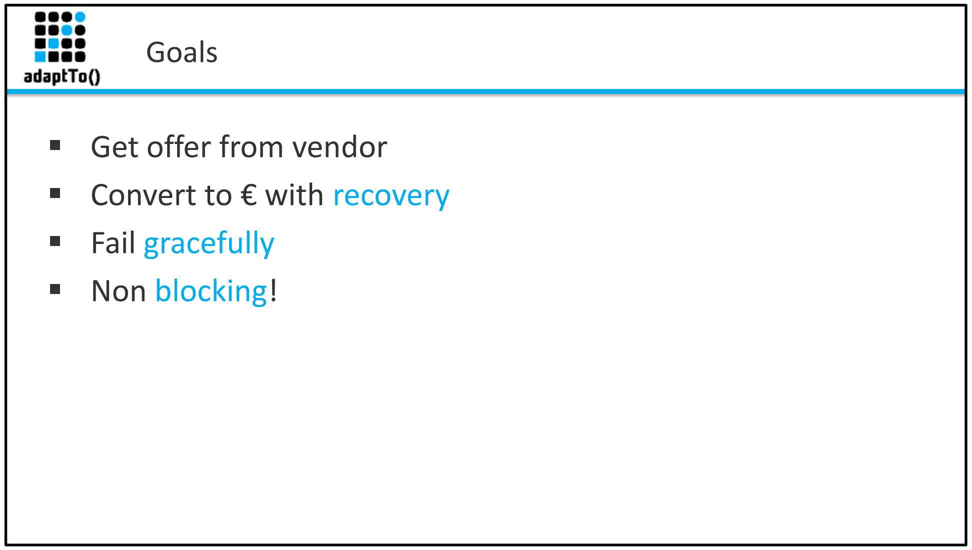
A vendor can be asked for an offer for an item. An offer contains a reference to the vendor, the item, a price and the currency.



Exchange Service

```
Promise<Offer> convertToEuro1(Offer offer);

Promise<Offer> convertToEuro2(Offer offer);
```

Two (encapsulations of) currency conversions services for converting an offer in a given currency to an offer in €. The two services might be able to handle different sets of currency. So we can fall back to the other one in case the first one fails. Note that the service might also fail due to other reasons (e.g. network partition), which also is good reason for falling back to the other service.

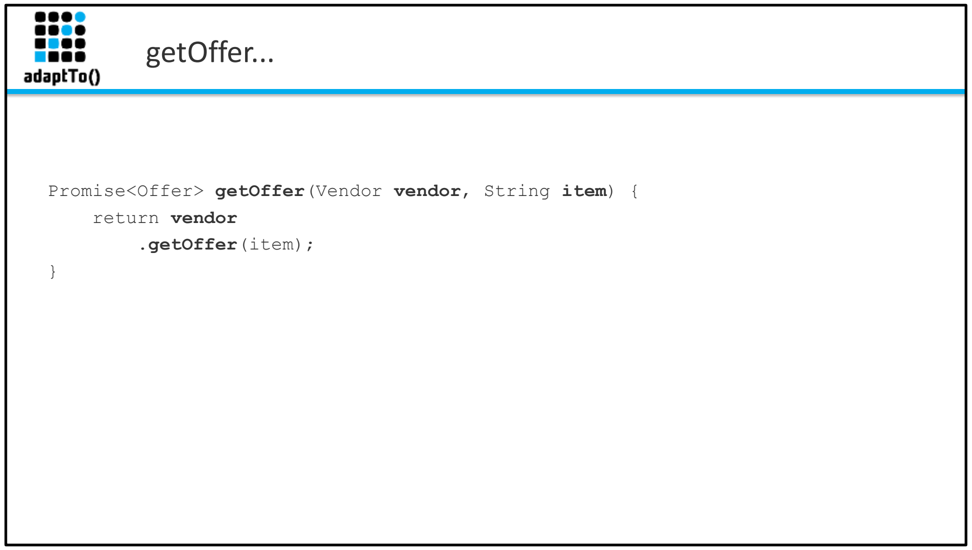


adaptTo()

Goals

- Get offer from vendor
- Convert to € with recovery
- Fail gracefully
- Non blocking!

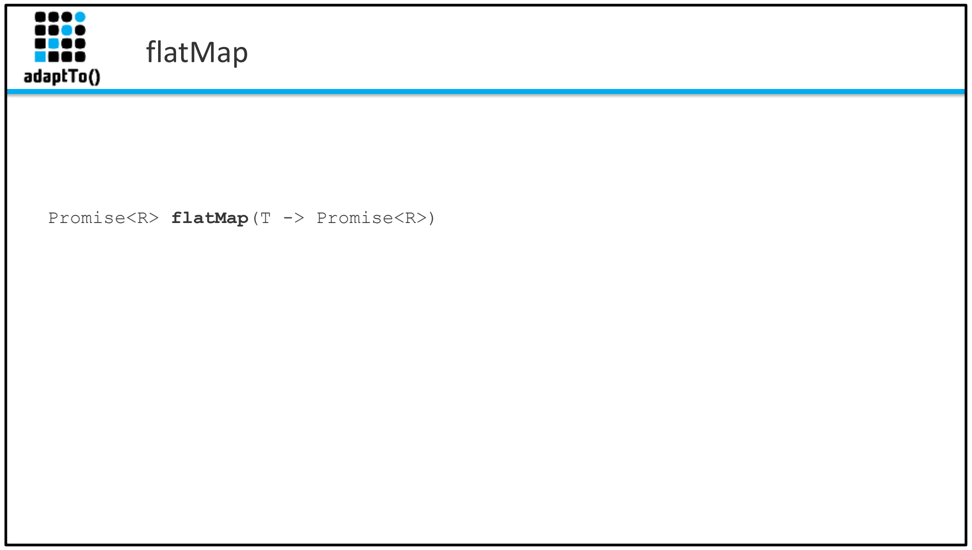
Get an offer from a vendor. The offer might be in any currency. Try to convert the offer to €. If this fails for any reason fall back to the other conversion service. Only if that one fails, fail the conversion for that item.
All of this should happen entirely non blocking and decoupled from hard coded thread management: it should be possible to execute this on a single worker thread without ever blocking that thread (i.e. that thread stays available for handing other work while e.g. waiting for the vendor service to come back with an offer).



getOffer...

```
Promise<Offer> getOffer(Vendor vendor, String item) {
    return vendor
        .getOffer(item);
}
```

Retrieve an offer for an item from the vendor. The offer might not be in the correct currency (€) though.

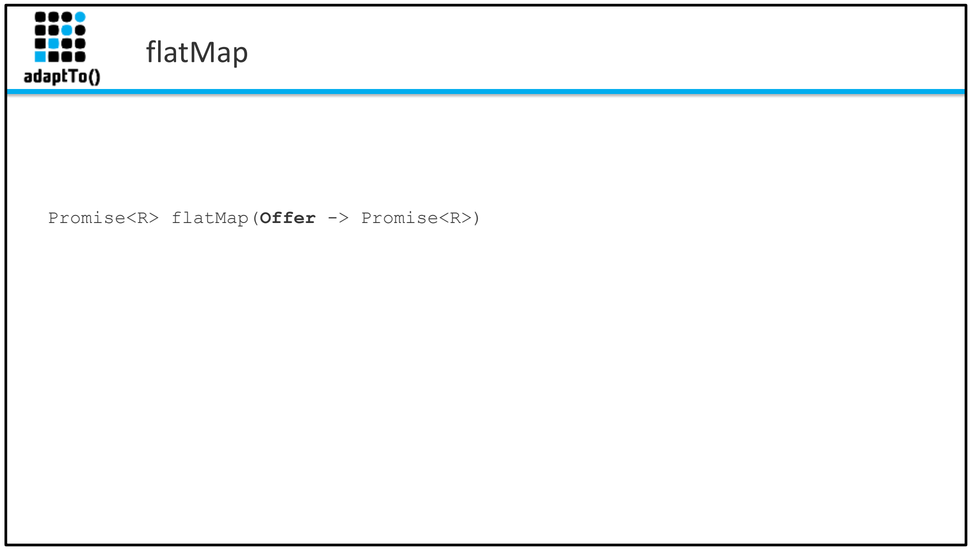


adaptTo()

flatMap

```
Promise<R> flatMap(T -> Promise<R>)
```

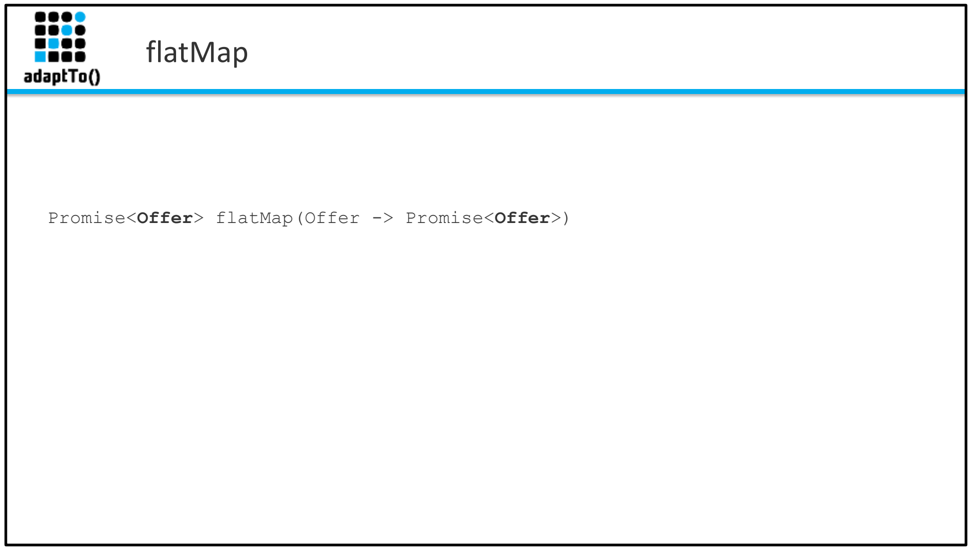
The flatMap combinator can be used for chaining promises. Given a lambda that maps T to Promise<R> it will return Promise<R>. The return promise will fail if this promise fails or if the promise returned by the lambda fails. It will resolve successfully if both promises resolve successfully.



flatMap

```
Promise<R> flatMap(**Offer** -> Promise<R>)
```

To see how we can use flatMap in conjunction with convertToEuro1 an approach is “to work out the types”. As we call flatMap on an instance of Promise<Offer>, T must be of type Offer.



adaptTo()

flatMap

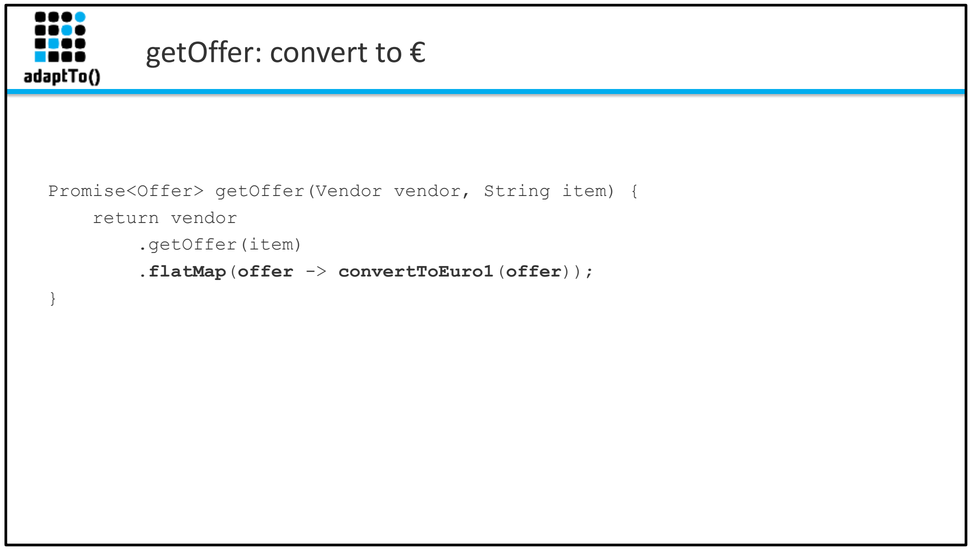
```
Promise<**Offer**> flatMap(Offer -> Promise<**Offer**>)
```

As converToEuro1 returns a Promise<Offer>, we want flatMap to return that type. The lambda we pass to flatMap thus also needs to return a Promise<Offer>.

flatMap

```
Promise<Offer> flatMap(Offer -> Promise<Offer>)

Promise<Offer> convertToEuro1(Offer);
```

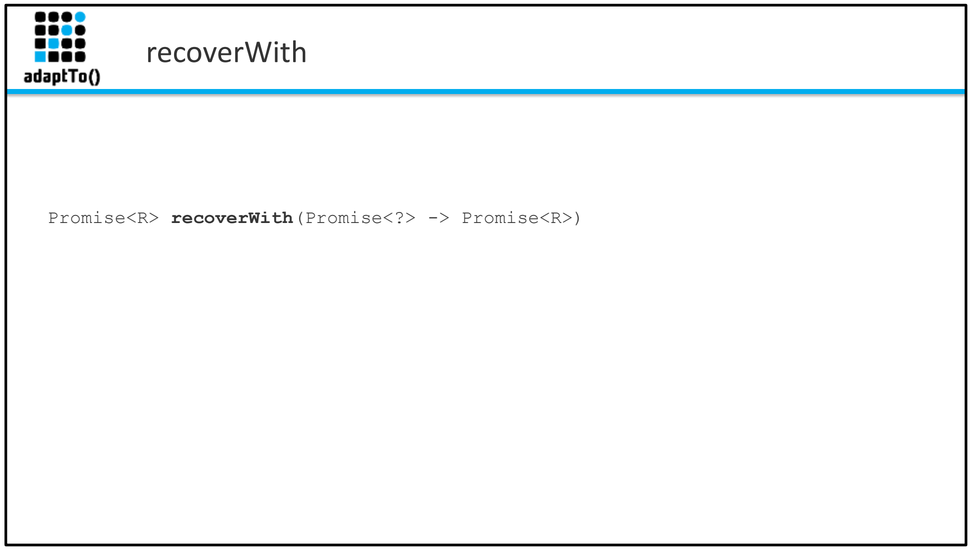
This matches the convertToEuro1 signature: it maps Offer to a Promise<Offer>.



getOffer: convert to €

```
Promise<Offer> getOffer(Vendor vendor, String item) {
    return vendor
        .getOffer(item)
        .flatMap(offer -> convertToEuro1(offer));
}
```

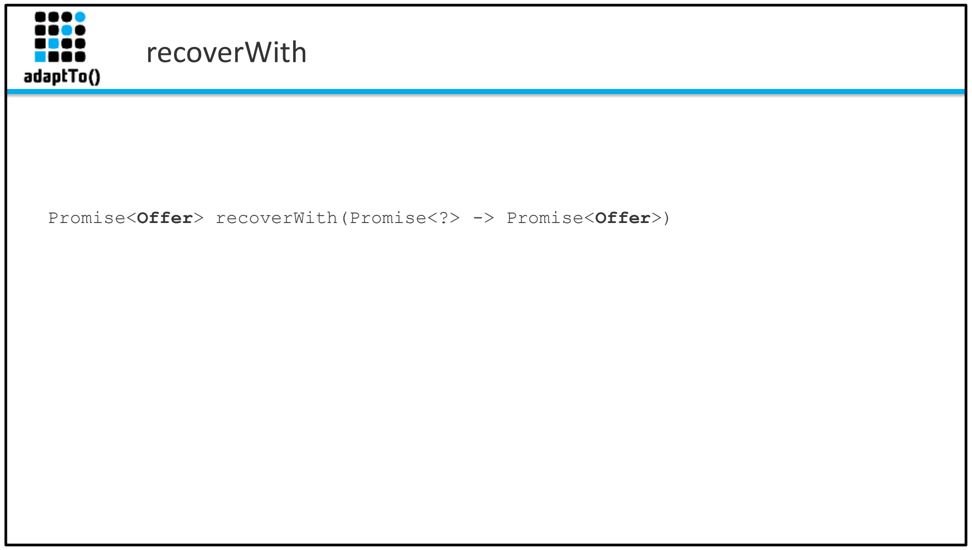
Pass a lambda constructed from the convertToEuro1 function (via eta conversion) to flatMap.
This version now does a first attempt at converting the offer to €. However it doesn’t fall back to a recovery service if conversion fails.



recoverWith

```
Promise<R> recoverWith(Promise<?> -> Promise<R>)
```

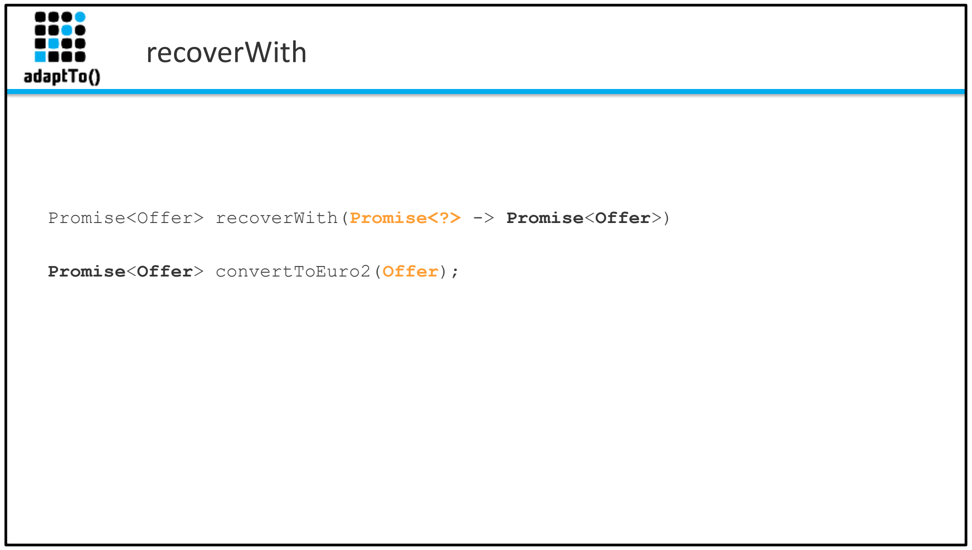
The recoverWith method can be used to recover a failed Promise with another Promise. The method takes a lambda that maps the failed promise to a recovery Promise.



recoverWith

```
Promise<Offer> recoverWith(Promise<?> -> Promise<Offer>)
```

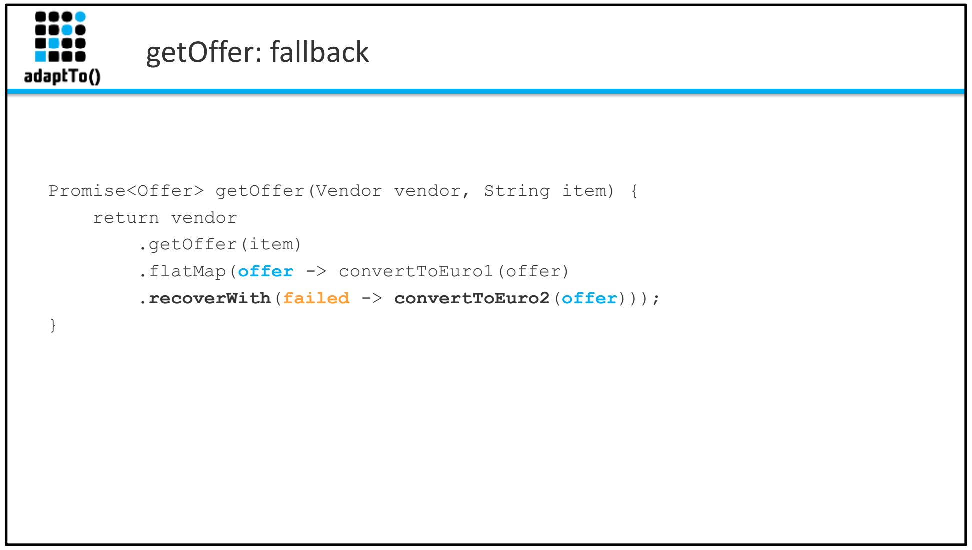
As we want a Promise<Offer> as return type, the lambda must also map to that type.



recoverWith

```
Promise<Offer> recoverWith(Promise<?> -> Promise<Offer>)

Promise<Offer> convertToEuro2(Offer);
```

It seems that we can’t construct a lambda using convertToEuro2 as we need to pass it an Offer but we only have a Promise<?>.

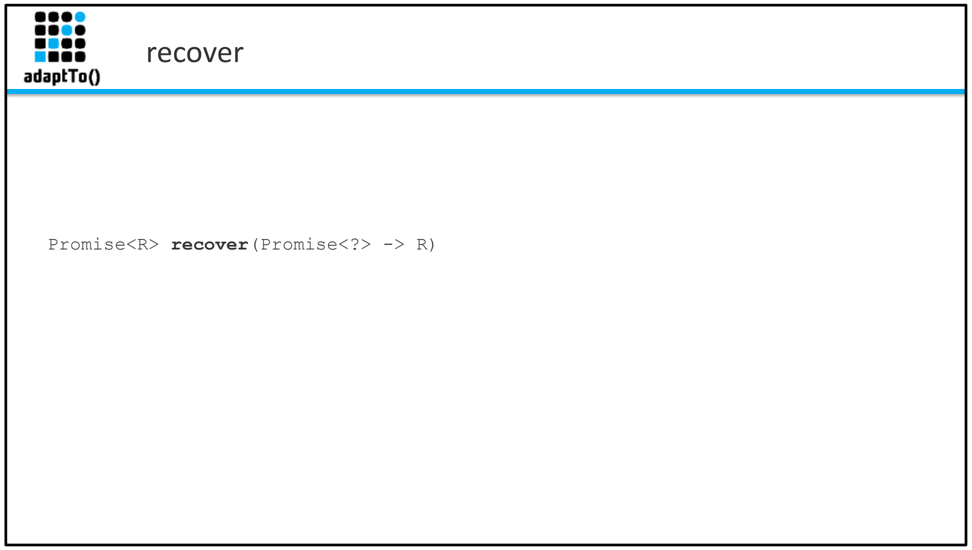


getOffer: fallback

```
Promise<Offer> getOffer(Vendor vendor, String item) {
    return vendor
        .getOffer(item)
        .flatMap(offer -> convertToEuro1(offer)
        .recoverWith(failed -> convertToEuro2(offer)));
}
```

Turns out that we can close over the initial offer and pass this one to converToEuro2 effectively ignoring the failed argument.

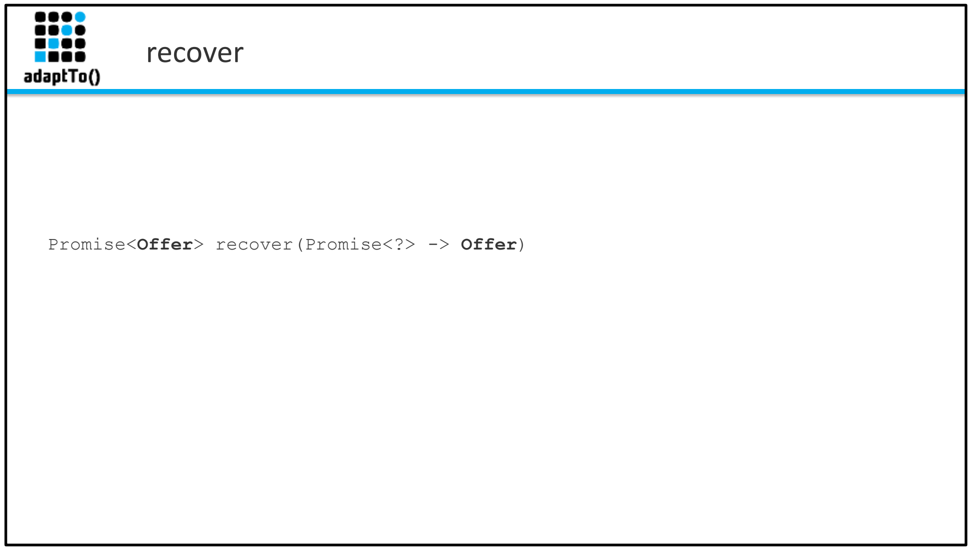
With this any failure encountered in convertToEuro1 will cause a fall back to convertToEuro2. Only when this also fails will the returned promise resolve with a failure. In order to make it fail gracefully we need to also recover from this.



recover

```
Promise<R> **recover**(Promise<?> -> R)
```

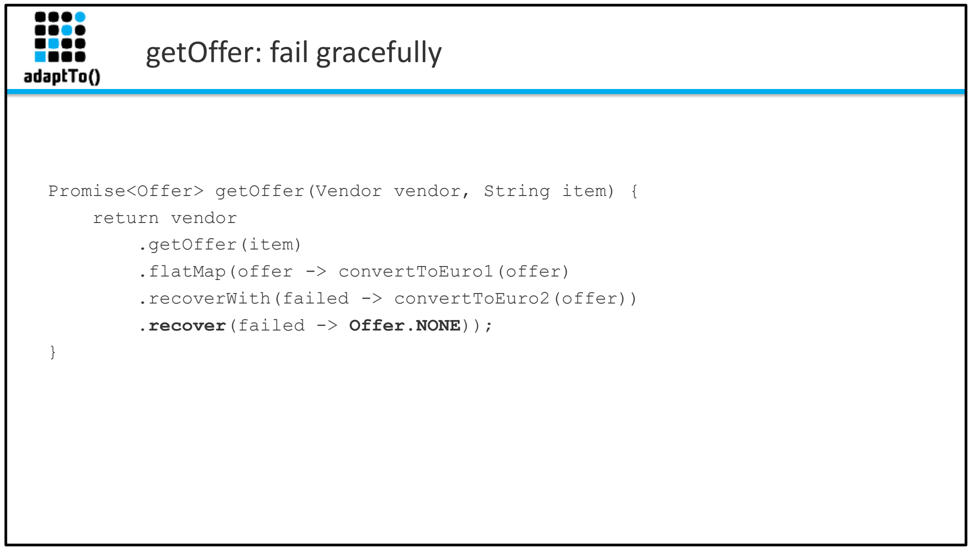
The recover method allows to recover a failed Promise with a value. The method takes a lambda that maps the failed promise to the value with which the return Promise will be resolved.



recover

```
Promise<Offer> recover(Promise<?> -> Offer)
```

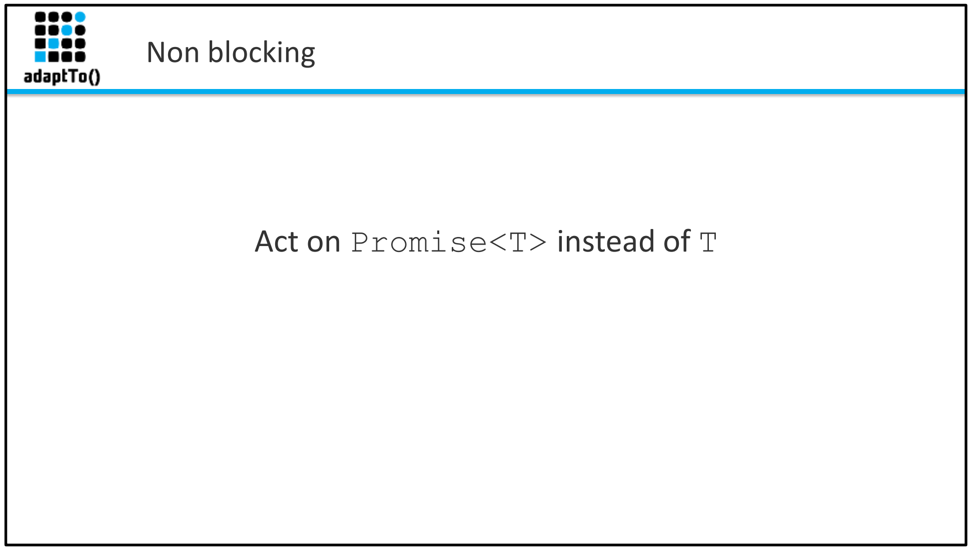
As we want to return an Promise<Offer> our lambda needs to return an Offer.



getOffer: fail gracefully

```
Promise<Offer> getOffer(Vendor vendor, String item) {
    return vendor
        .getOffer(item)
        .flatMap(offer -> convertToEuro1(offer)
        .recoverWith(failed -> convertToEuro2(offer))
        .recover(failed -> Offer.NONE));
}
```

We simple return a constant sentinel, which signals that no offer is available.
That sentinel will be returned either if getOffer fails or both conversions fail. The error is transparently chained through the calls keeping the code clean of error handling statements.



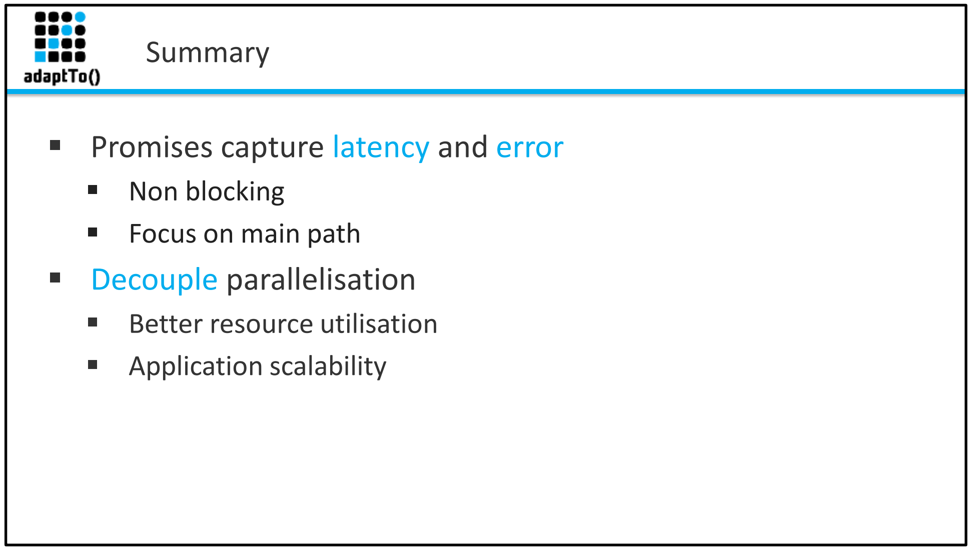
Non blocking

Act on `Promise<T>` instead of `T`

Lifting all functions into the Promise allows us to make the entire calculation non blocking. At the same time errors will be chained through keeping the focus on the main code path.

adaptTo()

Demo



adaptTo()

Summary

- Promises capture latency and error
  - Non blocking
  - Focus on main path
- Decouple parallelisation
  - Better resource utilisation
  - Application scalability

Making the effects of asynchronous calls explicit in the API signature (by returning a Promise<T> instead of T) allows us to

- cope with latency in an effective way regarding resource utilisation thereby maximising application throughput
- focusing on the error free path in the code while handling errors transparently.
- decouple parallelisation from threads leaving thread management to the environment / deployment
- Scale applications to run across a wide range of different hardware

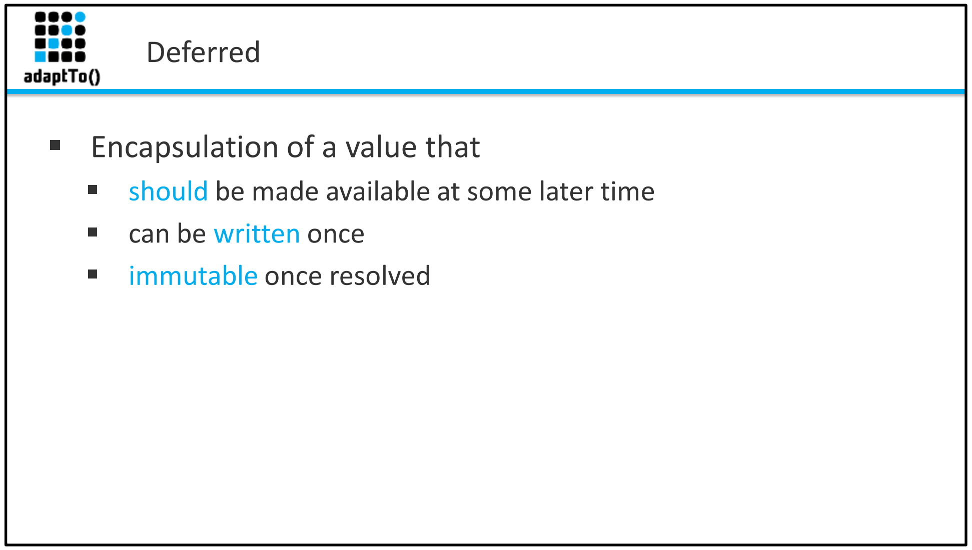
adaptTo()

Resources

http://goo.gl/pB9fza

adaptTo()

Appendix



Deferred

- Encapsulation of a value that
  - should be made available at some later time
  - can be written once
  - immutable once resolved

- Promise and Deferred: like the two sides of the same door, entrance and exit. The janitor opens the door (once) for the crowd to come it (many).
- Unfortunately there is some confusion regarding the nomenclature. The term Deferred is inherited from the Javascript world and refers to the same that in the Java world is usually called a Promise. But a Promise in our case is already what otherwise is called a Future…

What about j.u.concurrent.Future?

- Blocking semantics
  - No callbacks
- Not composable
  - No map/flatMap/filter/...

Resources

- Support material: https://github.com/mduerig/async-support/wiki
  - Session slides
  - Runnable demo
  - Links to further reading
- RFC 206: https://github.com/osgi/design/tree/master/rfcs/rfc0206
  - Public draft of OSGi RFC 206: Asynchronous Services
- OSGi Alliance: http://www.osgi.org/

Resources

- RxJava: https://github.com/ReactiveX/RxJava
  - Observables: taking Promises one step further
- Akka: http://akka.io/
  - Toolkit for concurrent and distributed applications on the JVM