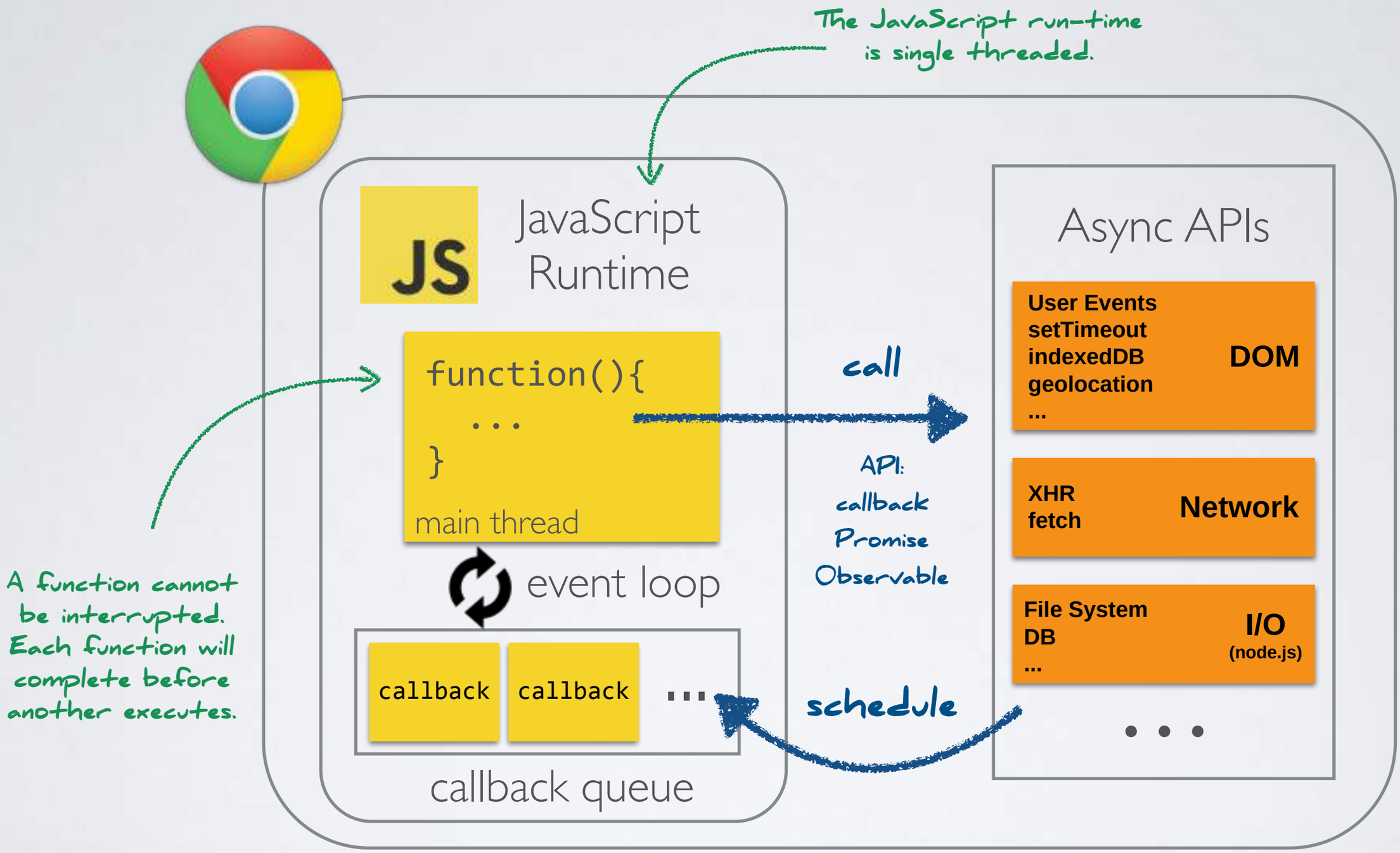
AJAX & Async JavaScript

Concurrency in JavaScript: Event Loop

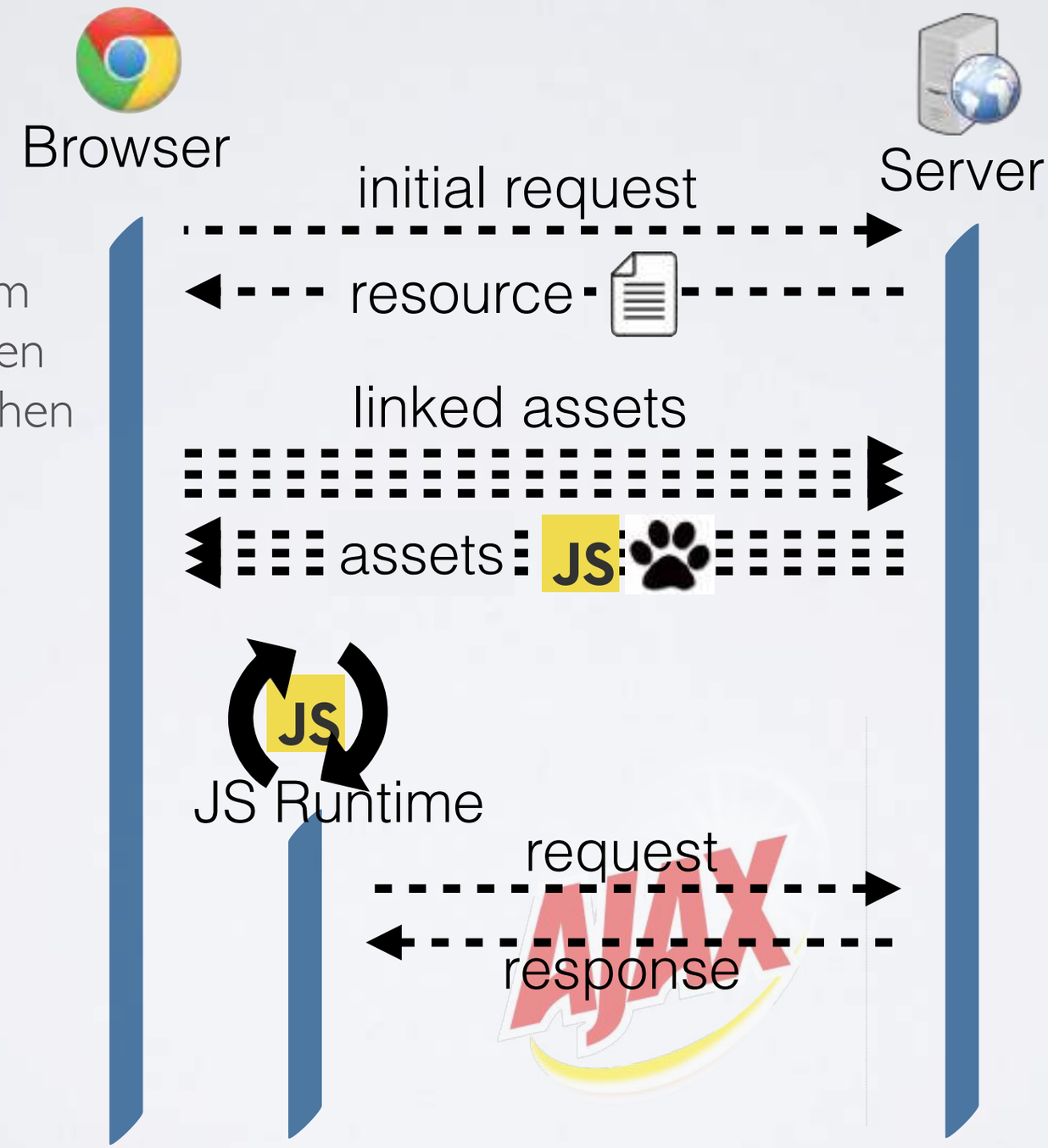


https://www.tedinski.com/2018/10/16/concurrency-vs-parallelism.html
What the heck is the event loop anyway? - https://www.youtube.com/watch?v=8aGhZQkoFbQ
Interactive Visualization: http://latentflip.com/loupe/
Jake Archibald: In The Loop: https://www.youtube.com/watch?v=cCOL7MC4Pl0
https://jakearchibald.com/2015/tasks-microtasks-queues-and-schedules/

Note: there is also the differentiation between macro- and micro tasks on the event loop: https://javascript.info/event-loop

Browsers & Network Access

Browsers can perform network requests when loading resources or when submitting forms.



The JavaScript runtime can perform network requests triggered by scripts.
These are called AJAX requests.

AJAX with Callbacks

using jQuery as an example

```
$.get('http://localhost:3001/comments', (data) => {
    console.log(data);
});
```

```
$.post('http://localhost:3001/comments',
      {text: 'test - ' + new Date()},
      () => {
         console.log('POST!');
});
```

EXERCISES

Exercise - AJAX with Callbacks

Promises

- A promise represents the result of an asynchronous operation.

Promises

- Promises are a way how to deal with asynchronity http://andyshora.com/promises-angularjs-explained-as-cartoon.html

One morning, a father says to his son: "Go and get the weather forecast, son!"

Promises

A promise represents the result of an asynchronous operation.

- Counterparts: Future<> in Java or Task<> in .NET (TPL)

Promises enable a programming model which is an alternative to callbacks.

- Asynchronous functions can return a value without blocking (so they look much more like synchronous functions)
- The return value is a promise that represents the final value that is possibly not yet known

Promises

Promises are a way to model asynchronous operations.
A promise is an object that represents the completion of an asynchronous operation and the resulting value.

Many different libraries implement promises for ES5.
(i.e. bluebird, Q, AngularJS, jQuery v3, ...)

https://promisesaplus.com/implementations

`Promise` is a built in object in ES2015.

```
const p = new Promise((resolve, reject) => {
   setTimeout(() => resolve('result), 1000);
});
p.then((value) => console.log(value));
p.catch((error) => console.log('Oh, crap!));
```

There are several polyfills for ES2015 promises.
i.e: https://github.com/taylorhakes/promise-polyfill

Using Promises

Consuming a promise:

```
promiseMeSomething()
    .then((value) => {
       // success handler
    })
    .catch((error) => {
      // error handler
    });
```

Provide a promise (EcmaScript 2015):

```
function promiseMeSomething(){
  const promise = new Promise((resolve, reject) => {
     // resolve the promise later
      setTimeout(function () { resolve("Success!");}, 1000);
  });
  return promise;
}
```

Promise Libraries

In ECMAScript 5 promises are provided by a library:

- bluebird: https://github.com/petkaantonov/bluebird
- Q: https://github.com/kriskowal/q
- when: https://github.com/cujojs/when
- RSVP.js: https://github.com/tildeio/rsvp.js
- Angular: https://docs.angularjs.org/api/ng/service/$q

obsolete

In ECMAScript 2015 promises are part of the language.
(as a consequence many new native browser APIs are using promises)

Polyfills: https://github.com/stefanpenner/es6-promise / https://github.com/taylorhakes/promise-polyfill

https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Promise

Evolution of ECMAScript Promises

ECMAScript 2015:

Construction:
- `new Promise((resolve, reject) => {})`
- `Promise.resolve(), Promise.reject()`

Handling/Flow:
- `Promise.prototype.then()`
- `Promise.prototype.catch()`

Combinators:
- `Promise.all()`
- `Promise.race()`

DOM API: Abort Controller (2017) -> can be used to cancel a Promise API

ECMAScript 2018: `Promise.prototype.finally()`

ECMAScript 2020: `Promise.allSettled()`

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/allSettled

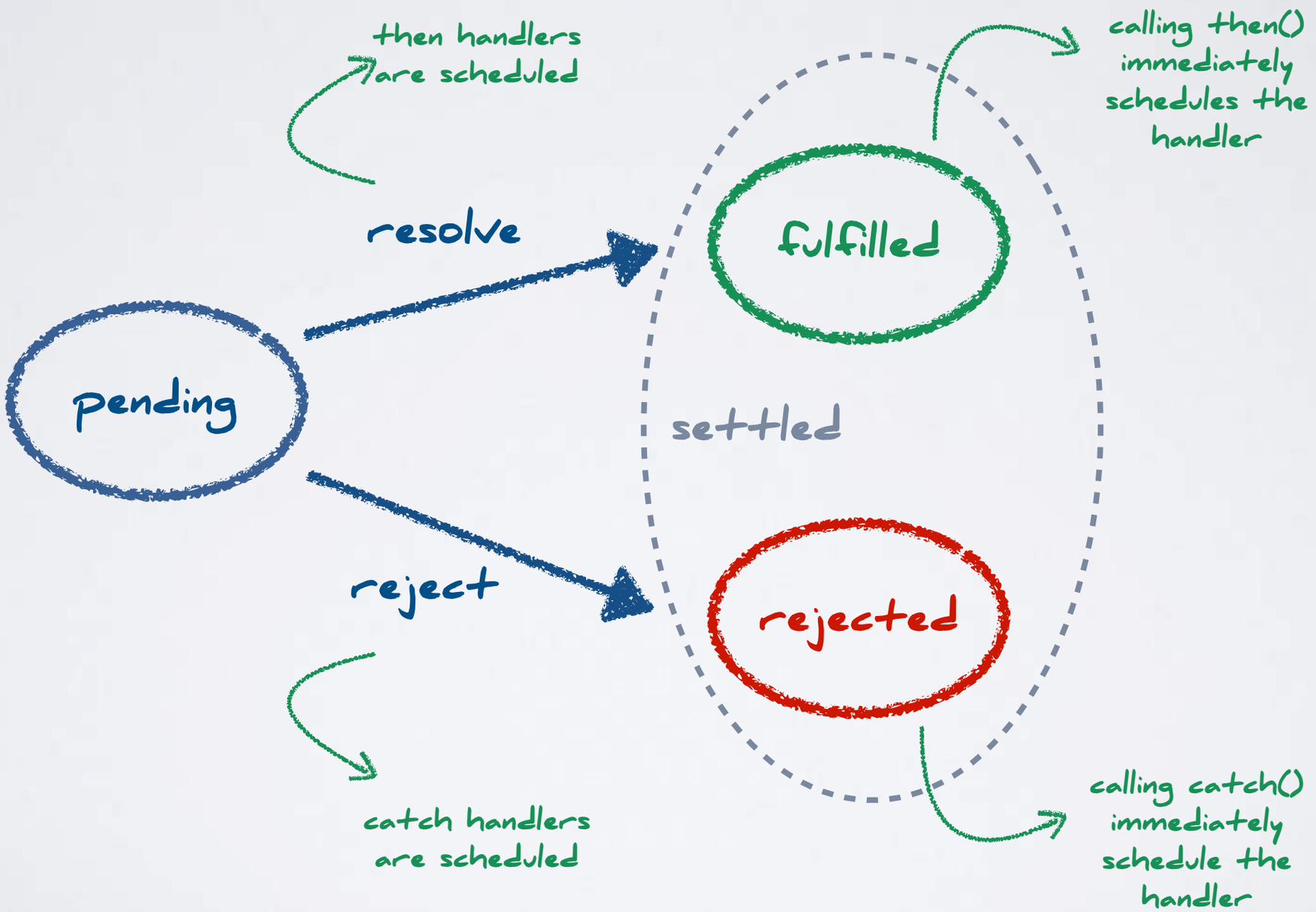
ECMAScript 2021: `Promise.any()`

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/any

ECMAScript 2024: `Promise.withResolvers()`

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/withResolvers

Promises represent a Statemachine



Advantages of Promises

- With promises the caller keeps in control over the program flow
- Promises decouple caller and executor of an asynchronous request.
  - Lower-level code can execute the request without having to know what has to be done when the request completes
  - Callers can (later) decide what should happen on completion

Advantages of Promises: Readability

Callback API:

```
promiseMeSomething(arg,() => {
  // success handler
},() => {
  // error handler
});
```

Promise API:

```
const promise = promiseMeSomething(arg);

promise.then((value) => {
  // success handler
});

promise.catch((value) => {
  // error handler
});
```

Advantages of Promises: Guaranteed Handler Execution

Traditional event handling:

```
const img1 = document.querySelector('.img-1');

img1.addEventListener('load', function() {
  // this will never be called if
  // the load event was emitted before the
  // handler was registerd
});
```

Promise API:

```
const promise = promiseMeSomething(arg);

promise.then((value) => {
  // handler is called even if
  // the promise was resolved before
  // the handler was registered
});
```

Advantages of Promises: Chaining aka Flattening the Pyramid of Doom

Advantages of Promises: Chaining Async Operations

Callback-Hell:

(aka Pyramid of Doom)

```
api((result) => {
    api2((result2) => {
        api3((result3) => {
            // do work
        });
    });
});
```

Promise Chaining:

If a "then-callback" returns another promise, the next "then" waits until it is settled.

```
api()
  .then((result) => {
    return api2();
  })
  .then((result2) => {
    return api3();
  })
  .then((result3) => {
     // do work
  })
  .catch(function(error) {
     //handle any error
});
```

```
api().then(api2)
     .then(api3)
     .then(/* do work */);
```

Advantages of Promises: Combining Multiple Actions

```
const firstData = null;
const secondData = null;

const responseCallback = () => {
 if (!firstData || !secondData) return;
 // do something
}

getData("http://url/first", (data) => {
  firstData = data;
  responseCallback();
});

getData("http://url/second", (data) => {
  secondData = data;
  responseCallback();
});
```

cumbersome orchestration

```
const promise1 = getData("http://url/first");
const promise2 = getData("http://url/second");

Promise.all([promise1, promise2])
    .then((arrayOfResults) => {…})
    .catch((errOfPromise) => {…});

Promise.race([promise1, promise2])
    .then((valOfPromise) => {…})
    .catch((errOfPromise) => {…});

Promise.any([promise1, promise2])
    .then((valOfPromise) => {…})
    .catch((errOfPromise) => {…});

Promise.allSettled([promise1, promise2])
    .then((valOfPromise) => {…})
    .catch((errOfPromise) => {…});
```

`Promise.all`: short-circuits when an input value is rejected

`Promise.race`: short-circuits when an input value is settled

`Promise.any`: short-circuits when an input value is fulfilled

`Promise.allSettled`: waits until all input valies are fulfilled or rejected

Promise Chaining

pay attention to the timing!

```
doSomething().then(function () {
  return doSomethingElse();
}).then(finalHandler);
```

```
doSomething().then(function () {
  doSomethingElse();
}).then(finalHandler);
```

```
doSomething()
  .then(doSomethingElse())
  .then(finalHandler);
```

```
doSomething()
  .then(doSomethingElse)
  .then(finalHandler);
```

https://pouchdb.com/2015/05/18/we-have-a-problem-with-promises.html

"Cancelling" a Promise API

using **AbortController** and **AbortSignal**

```
const controller = new AbortController();
const signal = controller.signal;
setTimeout(() => controller.abort(), 1000);

function promiseMeSomething(signal){
  if (signal.aborted) {
    return Promise.reject('aborted');
  }

  return new Promise((resolve, reject) => {
    setTimeout(() => resolve("Success!"), 2000);

    signal.addEventListener('abort', () => {
      reject('aborted!')
    });
  });
}

promiseMeSomething(signal)
  .then((value) => console.log(value))
  .catch((error) => console.log(error));
```

https://developer.mozilla.org/en-US/docs/Web/API/AbortSignal

AbortController is a DOM API: https://dom.spec.whatwg.org/#aborting-ongoing-activities

Promise.withResolvers()

resolve and reject is only available inside the Promise!

```
const promise = new Promise((resolve, reject) => {
   // resolve the promise from within
    setTimeout(function () { resolve("Success!");}, 1000);
});
// cannot resolve the promise here ...
```

```
const { promise, resolve, reject } = new Promise.withResolvers();

setTimeout(resolve('A value!'), 3000);
```

resolved from outside!

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/withResolvers

Promise-based Backend Access: `fetch` vs. `axios` vs. `ky`

`fetch` is a modern built-in browser API:

```
const res = await fetch('https://swapi.co/api/people/1/');
if (!res.ok) {
  throw new Error('Network response was not ok');
} else {
  const data = await res.json();
  console.log('SUCCESS', data);
}
```

non-2XX status is NOT an error!

`fetch` is available in all modern browsers (http://caniuse.com/#search=fetch).

For many projects, it makes sense to use a http library with more features:

- `ky`: https://github.com/sindresorhus/ky (modern, based on fetch)
- `axios`: https://github.com/axios/axios (legacy, based on XHR)

```
import ky from 'ky';
const response = await ky.get('http://www.example-api.com');
const data = await response.json<ResponseType>();
```

Features of http libraries: automatic json data transformation, status handling, http interceptors, response timeout, easily cancellable ...

EXERCISES

Exercise - AJAX with Promises

async / await

async / await

```
function sleep(millis){
  return new Promise((resolve, reject) => {
    setTimeout(() => resolve(), millis);
  });
}

async function start(){
  await sleep(1000);
  console.log('finished!');
}

start();
```

Note: `await` can be used inside an `async` function or on top-level

An `async` function (implicitly) returns a promise, which is resolved with the return value.

An `async` function can contain an `await` expression, that pauses the execution of the `async` function and waits for the passed `Promise`'s resolution, and then resumes the `async` function's execution and returns the resolved value.

The behavior of the event loop executing the code remains the same as in a promise chain.

`async` / `await` is a language feature of ES2017.
It is natively supported in all modern browsers.

https://caniuse.com/#feat=async-functions

async / await

`async`/`await` provides convenient language support on top of any Promise based API.

```
try {
   const response = await axios.get(API_URL);
   console.log(response);
} catch (error) {
   console.log(error)
}
```

```
const response = await axios.post(API_URL, payload);
console.log(response);
```

```
const response = await axios.delete(`${API_URL}/${id}`);
console.log(response);
```

Note: **`await`** can only be used inside an **`async`** function

(or on top level of modules, see https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await#top_level_await)

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function

async / await is still promises, but with a really nice syntax!

```
function sleep(millis){
  return new Promise((resolve, reject) => {
    setTimeout(() => resolve(), millis);
  });
}

async function start(){
  await sleep(1000).then(() => console.log('waking up!'));
  console.log('finished!');
}

start();
```

The behavior of the event loop executing the code remains the same as in a promise chain.

Async Generators & `for await...of`

```
let count = 0;
function sleep(millis) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(++count);
    }, millis);
  });
}

async function* start() {
  yield sleep(2000);
  yield sleep(2000);
  yield 42;
}
```

```
async function doWork() {
  let iterator = start();
  console.log('1');
  const val1 = await iterator.next();
  console.log('Val 1', val1);
  const val2 = await iterator.next();
  console.log('Val 2', val2);
  const val3 = await iterator.next();
  console.log('Val 3', val2);

  for await (let val of start()) {
    console.log('Val', val);
  }
}
doWork();
```

Warning: be aware of unhandled promise rejections: https://jakearchibald.com/2023/unhandled-rejections/

EXERCISES

Exercise - Async / Await

EXERCISES

Exercise - Number-Guess

RxJS / Observables

Observables

An observable represents a multi-value push protocol:

	Single	Multiple
Pull	`Function`	Iterator
Push	`Promise`	`Observable`

Typical scenario: an asynchronous stream of events.

```
const observable =
   rxjs.fromEvent(
      document.getElementById('my-button'),'click'
   );

observable.subscribe(x => console.log(x));
```

Observables in ECMAScript

RxJS is the "de-facto" implementation of observables today.

RxJS

https://rxjs.dev/ ??? -> https://rxjs-dev.firebaseapp.com

RxJS is a library for composing asynchronous and event-based programs by using observable sequences.

There is a proposal to standardize **`Observable`** as built-in type in a future version of ECMAScript.

There are different competing libraries with similar reactive programming concepts: Beacon.js, xstream, …

https://github.com/tc39/proposal-observable

http://reactivex.io/
https://baconjs.github.io/
http://staltz.com/xstream/

What is RxJS?

Reactive programming is an asynchronous programming paradigm concerned with data streams and the propagation of change (Wikipedia).

RxJS (Reactive Extensions for JavaScript) is a library for reactive programming using observables that makes it easier to compose asynchronous or callback-based code.

RxJS is a library with an API. It makes it possible to model reactive programming constructs *explicitly in your code*:
Observables, Observers, Subjects, Subscriptions ...

https://angular.io/guide/rx-library

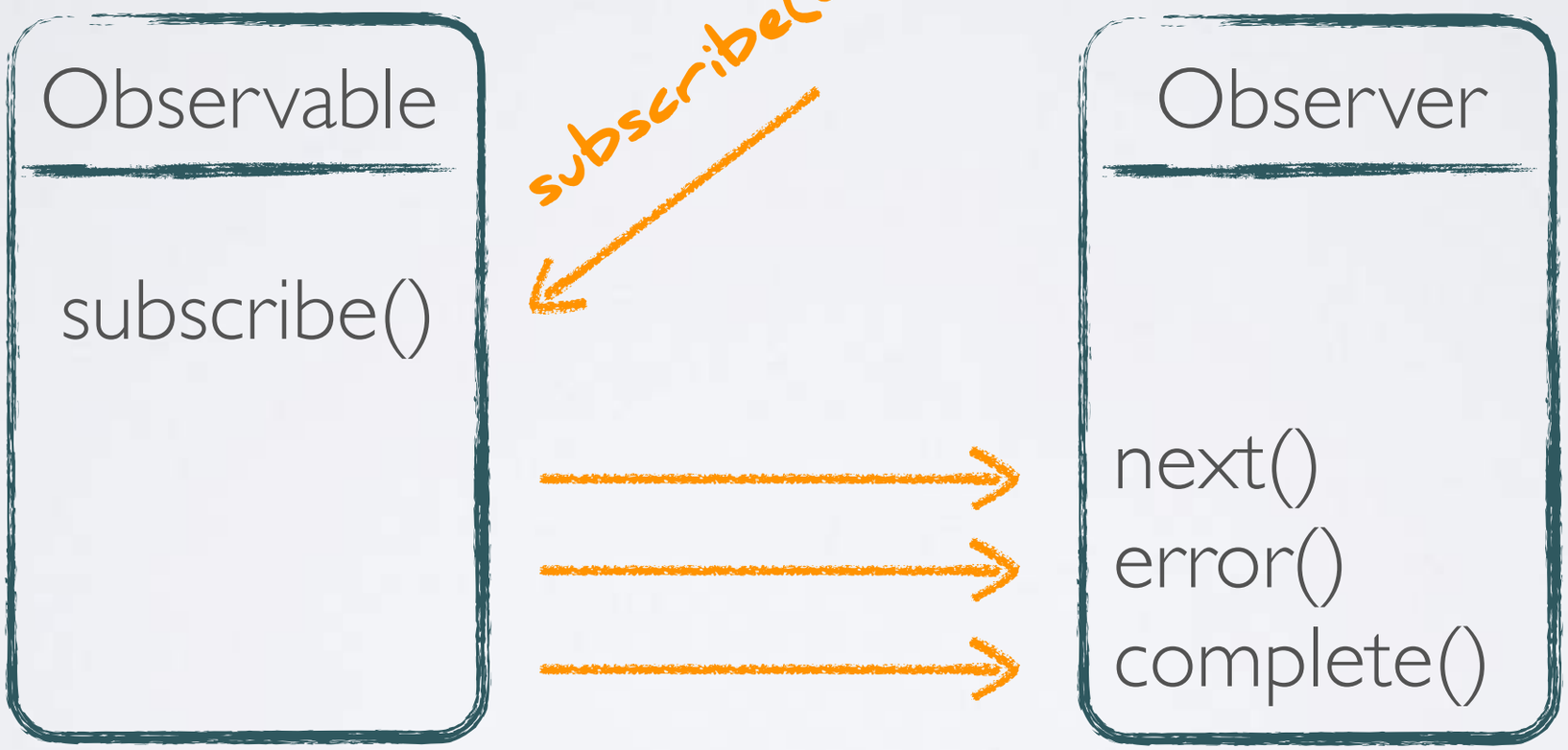
Observables

Characteristics of Observables

- Represents any number of values over any amount of time
- Observables are able to deliver values either synchronously or asynchronously
- Observables are lazy
- Observables can be cancelled
- Observables can be composed with higher-order combinators

RxJS:
"The Observer pattern done right"

http://reactivex.io/



The power of RxJS are the operators on observables and the composability of observables.

Using Observables

Consume an observable:

```
var value$ = startAsyncOperation();
value$.subscribe(
  value => document.getElementById("content").innerText += value,
  error => console.log('Error: ' + error),
  () => console.log('Completed!')
);
```

Provide an observable (using RxJS):

```
function startAsyncOperation(){
   var value$ = new rxjs.Observable(
     observer => {
       setTimeout(() => observer.next("This is first value!"), 1000);
       setTimeout(() => observer.next("This is second value!"), 2000);
       setTimeout(() => observer.next("This is third value!"), 3000);
       setTimeout(() => observer.complete(), 4000);
     });
   return value$;
}
```

Note: In typical application programming you *never* have to create an observable like this.
You either use factory functions or you consume an API that exposes an "event source" as an observable.

Observables vs. Promises

An observable can be an alternative to a promise:
a stream that pushes exactly one result.

```
const observable = new rxjs.Observable(
   observer => setTimeout(
        () => {
             observer.next(42);
             observer.complete();
        },
        2000);
);

observable.subscribe(x => console.log(x));
```

Angular exposes observables for backend access via `HttpClient` service.

Promises vs. Observables

Promise	**Observable**
returns a single value	works with multiple values over time
cancellable via `AbortController`	cancellable
enables the `async/await` syntax of ES2017	supports powerful operators like map, filter & reduce ("lo-dash for async")

For single AJAX calls observables do not provide a clear advantage.
But observables can be used consistently for all kind of async operations.
The power of onservables unfolds when composing several async operations.

https://developer.mozilla.org/en-US/docs/Web/API/AbortController
https://developers.google.com/web/updates/2017/09/abortable-fetch
http://blog.thoughtram.io/angular/2016/01/06/taking-advantage-of-observables-in-angular2.html
Do Observables make sense for http? - https://github.com/angular/angular/issues/5876

Sync vs. Async Observables

sync:

```
of(1, 2, 3, 4, 5)
  .subscribe(
    value => console.log(value),
  );
console.log('Done 1');
```

async:

```
interval(1000)
  .subscribe(
    value => console.log(value),
  );
console.log('Done 2');
```

Creating Observable Streams

Creation Functions:

```
of, from, fromEvent,
interval, timer ...
```

Manual Creation:

```
new Observable(observer => {
  observer.next(1);
  observer.next(2);
  observer.complete();
});
```

```
new Observable(observer => {
  setTimeout(() =>  observer.next(1), 1000);
  setTimeout(() =>  observer.next(2), 2000);
  setTimeout(() =>  observer.complete(), 3000);
});
```

Note: In typical application programming you *never* have to manually create an observable.
You either use creation functions or you consume an API that exposes an "event source" in an observable.

Subscription & Unsubscribe

A subscription is modeled by a **`Subscription`** object.

```
const subscription = interval(1000)
  .subscribe(
    value => console.log(value),
    error => console.log(error),
    () => console.log('COMPLETED')
  );

setTimeout(() => {
  console.log(subscription.closed);
  subscription.unsubscribe();
}, 4000)
```

There is a potential memory leak, if you are not unsubscribing from an Observable!

Cold vs. Hot Observables

A cold observable creates the producer:

```
var cold = new Observable((observer) => {
  var producer = new Producer();
  // have observer listen to producer here
});
```

Each subscription creates it's own producer. The producer only starts producing with the subscription.

A hot observable closes over the producer:

```
var producer = new Producer();
var hot = new Observable((observer) => {
  // have observer listen to producer here
});
```

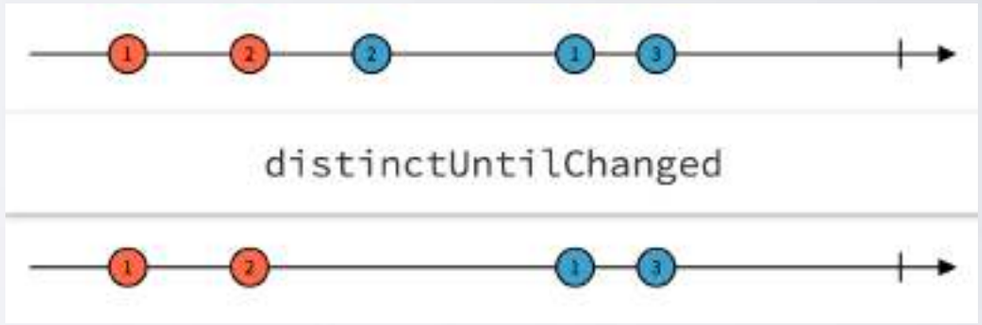
The producer produces values independent of subscriptions.

https://medium.com/@benlesh/hot-vs-cold-observables-f8094ed53339

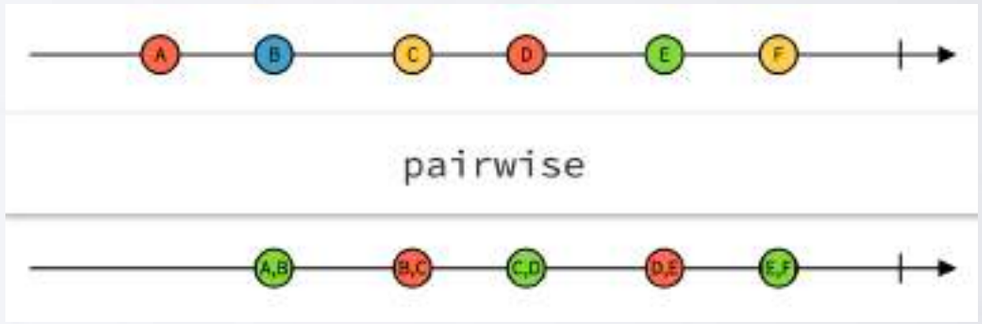
Transformation Operators

```
const {filter, map} = rxjs.operators;
const o1 = rxjs
             .interval(1000)
             .pipe(
                  filter(v => v % 2 === 0),
                  map(v => v * 2)
              );
```

```
const {distinctUntilChanged} = rxjs.operators;
const transformed = o1.
             .pipe(
                distinctUntilChanged()
             );
```




```
const {pairwise} = rxjs.operators;
const transformed = o1
             .pipe(
                pairwise()
             );
```




```
transformed.subscribe(
  v => console.log(v)
);
```

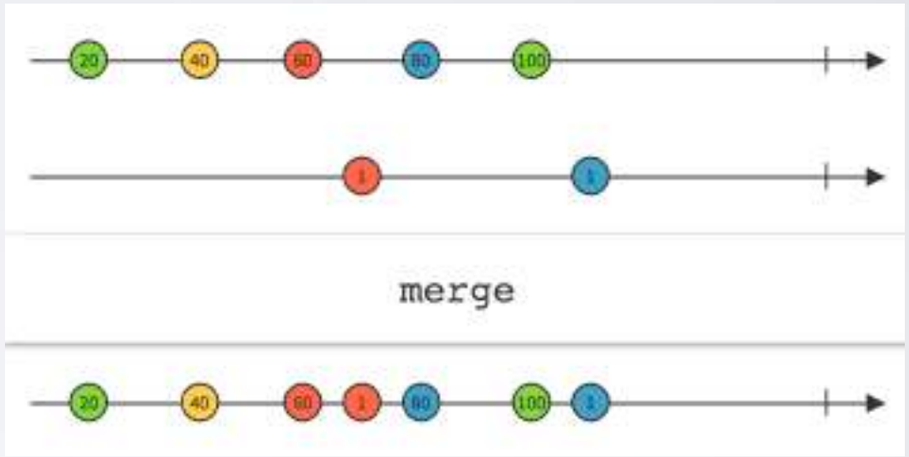
https://rxjs.dev/guide/operators
http://rxmarbles.com/

Combination Operators

```
const o1 = rxjs.interval(1000);
const o2 = rxjs.interval(1000)
  .delay(500)
  .map(v => v * 1000);
```

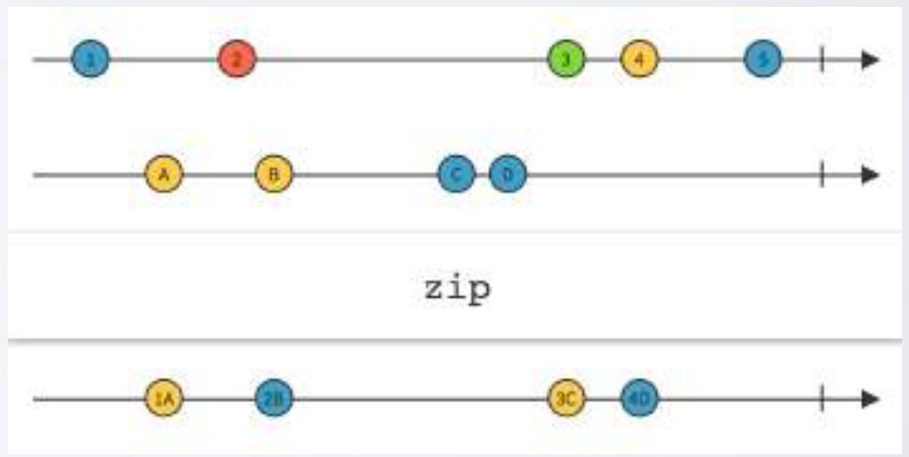
```
const combined = rxjs.merge(o1,o2);

combined.subscribe(
  v => console.log(v)
);
```




```
const combined = rxjs.zip(o1,o2);

combined.subscribe(
  v => console.log(v)
);
```



http://reactivex.io/documentation/operators
http://rxmarbles.com/

Higher Order Observables

"Observables of Observables"

```
function api1(){
  return rxjs.of(42).pipe(delay(2000));
}
function api2(){
  return rxjs.of(43).pipe(delay(1000));
}
function api3(){
  return rxjs.of(44).pipe(delay(500));
}

stream = rxjs.from([api1, api2, api3]);
```

```
stream
  .pipe(mergeMap(x => x()))
  .subscribe(console.log);
```

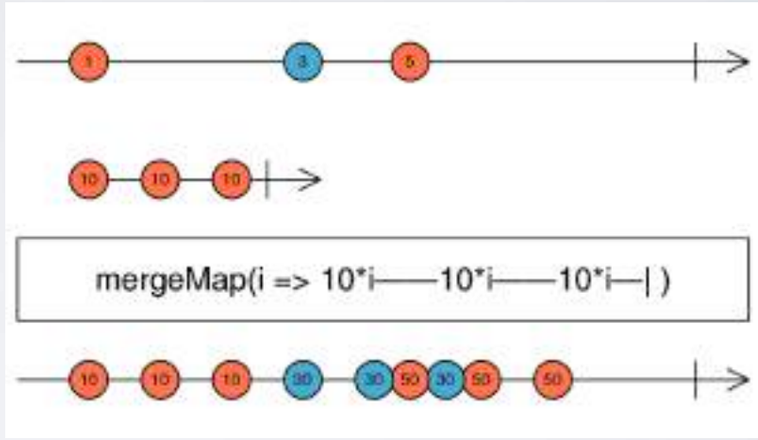
out:
44, 43, 42

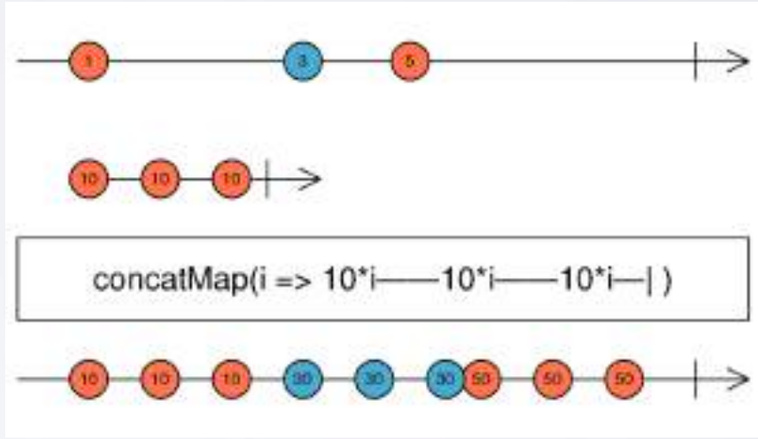
```
stream
  .pipe(concatMap(x => x()))
  .subscribe(console.log);
```

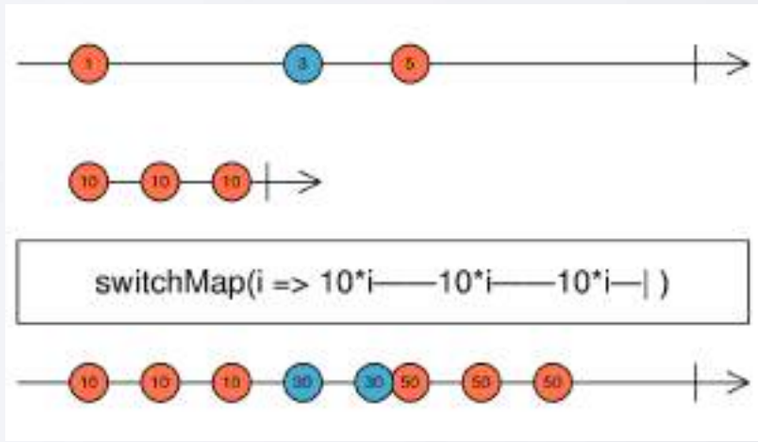
out:
42, 43, 44

```
stream
  .pipe(switchMap(x => x()))
  .subscribe(console.log);
```

out:
44





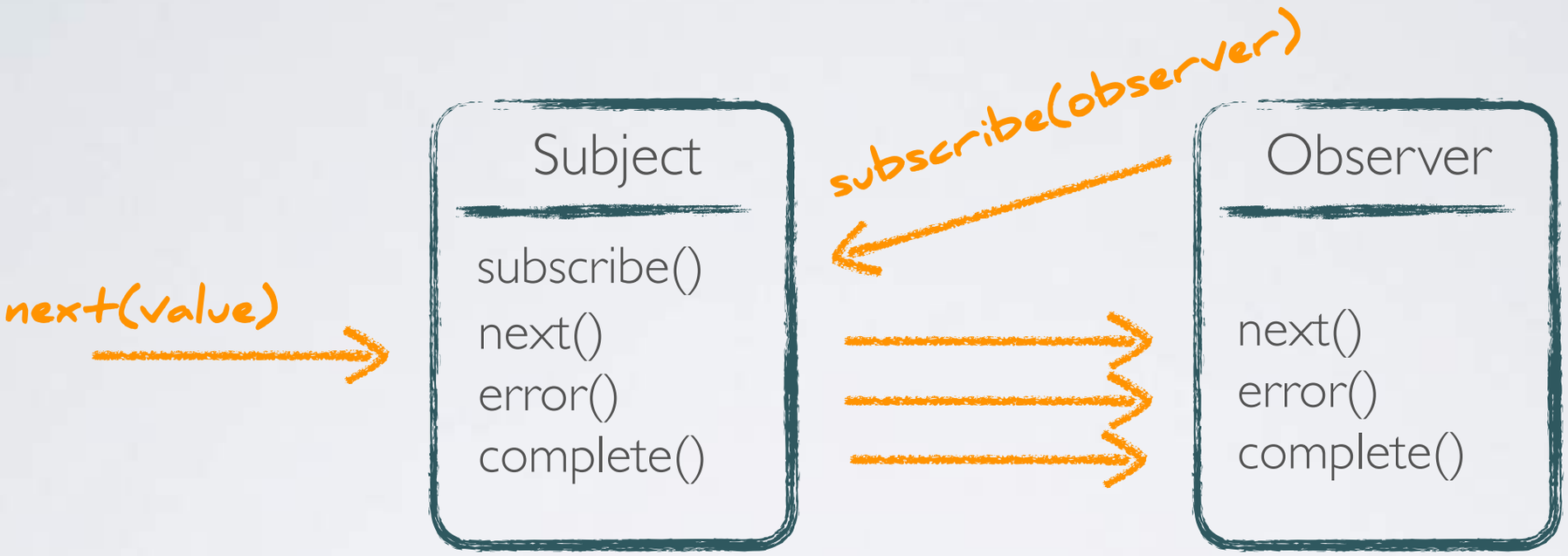


Demo:

Type-Ahead Wikipedia Search

Subject

A `Subject` is a `Observable` and a `Observer` at the same time.



Subjects can be used as "manually triggered" streams.
Subjects can be used for multicasting to several Observers.

```
const subject = new rxjs.Subject();

subject.subscribe(
  value => console.log(value),
);

subject.next(43);
```

Hint: *Try to avoid Subjects*. They are an (imperative) indirection that introduces state and breaks the flow between producer and consumer. Try to use declarative (transformed) event streams from producers to consumers.

BehaviorSubject

A `BehaviorSubject` is a special `Subject`, which has a notion of "the current value". It stores the latest value emitted to its consumers, and whenever a new `Observer` subscribes, it will immediately receive the "current value" from the `BehaviorSubject`.

`BehaviorSubjects` are useful for representing "values over time". For instance, an *event stream of birthdays* is a `Subject`, but the *stream of a person's age* would be a `BehaviorSubject`.

```
const subject = new rxjs.BehaviorSubject(42);

subject.subscribe(
  value => console.log(value),
);

subject.next(43);
```

Other variants of `Subject`: `AsyncSubject`, `ReplaySubject`

EXERCISES

Exercise 3 - AJAX with RxJS