Classes:

A Class User{...}, creates a function named User and stores it's methods in User.prototype

We cannot call a class without the new keyword.

```
class User {
  constructor() {}
}

alert(typeof User); // function
User(); // Error: Class constructor User cannot be invoked without 'ne
```

If the class expression has a name, it's visible inside it's class only.

```
// "Named Class Expression"
// (no such term in the spec, but that's similar to Named Function Expres
let User = class MyClass {
  sayHi() {
    alert(MyClass); // MyClass name is visible only inside the class
  }
};

new User().sayHi(); // works, shows MyClass definition

alert(MyClass); // error, MyClass name isn't visible outside of the class
```

The problem is called "losing `this`".

There are two approaches to fixing it, as discussed in the chapter Function binding:

1. Pass a wrapper-function, such as `setTimeout(() => button.click(), 1000)`.
2. Bind the method to object, e.g. in the constructor.

Class fields provide another, quite elegant syntax:

```
class Button {
  constructor(value) {
    this.value = value;
  }
  click = () => {
    alert(this.value);
  }
}

let button = new Button("hello");

setTimeout(button.click, 1000); // hello
```

The class field `click = () => {...}` is created on a per-object basis, there's a separate function for each `Button` object, with `this` inside it referencing that object. We can pass `button.click` around anywhere, and the value of `this` will always be correct.

Class Inheritance:

Any expression is allowed after `extends`

Class syntax allows to specify not just a class, but any expression after `extends`.

For instance, a function call that generates the parent class:

```
function f(phrase) {
  return class {
    sayHi() { alert(phrase); }
  };
}

class User extends f("Hello") {}

new User().sayHi(); // Hello
```

Here `class User` inherits from the result of `f("Hello")`.

That may be useful for advanced programming patterns when we use functions to generate classes depending on many conditions and can inherit from them.

Super:

Super.method() to call a parent method

Super() to call a parent constructor

The short answer is:

- **Constructors in inheriting classes must call `super(...)`, and (!) do it before using `this`.**

…But why? What's going on here? Indeed, the requirement seems strange.

Of course, there's an explanation. Let's get into details, so you'll really understand what's going on.

In JavaScript, there's a distinction between a constructor function of an inheriting class (so-called "derived constructor") and other functions. A derived constructor has a special internal property `[[ConstructorKind]]:"derived"`. That's a special internal label.

That label affects its behavior with `new`.

- When a regular function is executed with `new`, it creates an empty object and assigns it to `this`.
- But when a derived constructor runs, it doesn't do this. It expects the parent constructor to do this job.

So a derived constructor must call `super` in order to execute its parent (base) constructor, otherwise the object for `this` won't be created. And we'll get an error.

For the `Rabbit` constructor to work, it needs to call `super()` before using `this`, like here:

The parent constructor ALWAYS uses it's own field values, not the overridden(child's) fields.

Please note: now the output is different.

And that's what we naturally expect. When the parent constructor is called in the derived class, it uses the overridden method.

…But for class fields it's not so. As said, the parent constructor always uses the parent field.

Why is there a difference?

Well, the reason is the field initialization order. The class field is initialized:

- Before constructor for the base class (that doesn't extend anything),
- Immediately after `super()` for the derived class.

In our case, `Rabbit` is the derived class. There's no `constructor()` in it. As said previously, that's the same as if there was an empty constructor with only `super(...args)`.

So, `new Rabbit()` calls `super()`, thus executing the parent constructor, and (per the rule for derived classes) only after that its class fields are initialized. At the time of the parent constructor execution, there are no `Rabbit` class fields yet, that's why `Animal` fields are used.

This subtle difference between fields and methods is specific to JavaScript.

Luckily, this behavior only reveals itself if an overridden field is used in the parent constructor. Then it may be difficult to understand what's going on, so we're explaining it here.

If it becomes a problem, one can fix it by using methods or getters/setters instead of fields.

[[HomeObject]] stores the object when the function or method is called

```
let animal = {
  sayHi() {
    alert(`I'm an animal`);
  }
};

// rabbit inherits from animal
let rabbit = {
  __proto__: animal,
  sayHi() {
    super.sayHi();
  }
};

let plant = {
  sayHi() {
    alert("I'm a plant");
  }
};

// tree inherits from plant
let tree = {
  __proto__: plant,
  sayHi: rabbit.sayHi // (*)
};

tree.sayHi();  // I'm an animal (?!?)
```

Methods, not function properties

`[[HomeObject]]` is defined for methods both in classes and in plain objects. But for objects, methods must be specified exactly as `method()`, not as `"method: function()"`.

The difference may be non-essential for us, but it's important for JavaScript.

In the example below a non-method syntax is used for comparison. `[[HomeObject]]` property is not set and the inheritance doesn't work:

```
let animal = {
  eat: function() { // intentionally writing like this instead of eat() {...
    // ...
  }
};

let rabbit = {
  __proto__: animal,
  eat: function() {
    super.eat();
  }
};

rabbit.eat();  // Error calling super (because there's no [[HomeObject]])
```

Arrow functions have no `super`

As was mentioned in the chapter Arrow functions revisited, arrow functions do not have `super`.

If accessed, it's taken from the outer function. For instance:

```
class Rabbit extends Animal {
  stop() {
    setTimeout(() => super.stop(), 1000); // call parent stop after 1sec
  }
}
```

The `super` in the arrow function is the same as in `stop()`, so it works as intended. If we specified a "regular" function here, there would be an error:

```
// Unexpected super
setTimeout(function() { super.stop() }, 1000);
```

Prototype:

[[Prototype]] is a property which stores references to another object. The obj that is being referenced is called prototype.

Limitations:

1. Can't have Circular prototype references, JS throws error.
2. Value of proto can only be null or an obj.

__proto__ is the getter/setter for prototype and not the prototype itself

This is not at all affected by prototypes, this always points to the object before dot.

`F.prototype` only used at `new F` time

`F.prototype` property is only used when `new F` is called, it assigns `[[Prototype]]` of the new object.

If, after the creation, `F.prototype` property changes (`F.prototype = <another object>`), then new objects created by `new F` will have another object as `[[Prototype]]`, but already existing objects keep the old one.

Default F.prototype, constructor property

Every function has the `"prototype"` property even if we don't supply it.

The default `"prototype"` is an object with the only property `constructor` that points back to the function itself.

Like this:

```
function Rabbit() {}

/* default prototype
Rabbit.prototype = { constructor: Rabbit };
*/
```

Obj.prototype sets up the [[Prototype]] for new objs After that line, but Not changes the proto of objs Before it.

Delete and Insert operation only works for the Given Obj. It Does NOT change the parent properties.

Null and undefined does not have it's own wrapper objects.

We use static to declare a property for the Class and NOT it's objects. Hence, we cannot access the property using obj.method() but only with Class.method()

Can declare private and protected properties in the JS by using #fieldName or _fieldName respectively. Use the getter and setter to change it's values.

Please note a very interesting thing. Built-in methods like `filter`, `map` and others – return new objects of exactly the inherited type `PowerArray`. Their internal implementation uses the object's `constructor` property for that.

In the example above,

```
arr.constructor === PowerArray
```

When `arr.filter()` is called, it internally creates the new array of results using exactly `arr.constructor`, not basic `Array`. That's actually very cool, because we can keep using `PowerArray` methods further on the result.

Even more, we can customize that behavior.

We can add a special static getter `Symbol.species` to the class. If it exists, it should return the constructor that JavaScript will use internally to create new entities in `map`, `filter` and so on.

If we'd like built-in methods like `map` or `filter` to return regular arrays, we can return `Array` in `Symbol.species`, like here:

Built in objects like Arr, Obj do not inherit static methods and objs.

	works for	returns
`typeof`	primitives	string
`{}.toString`	primitives, built-in objects, objects with `Symbol.toStringTag`	string
`instanceof`	objects	true/false

Mixins

Mixins class is simply copied to the prototype of object.

Maps:

Maps stores the key value pairs like objs but unlike it, we can have keys of any types in it, even obj as keys.

How `Map` compares keys

To test keys for equivalence, `Map` uses the algorithm SameValueZero. It is roughly the same as strict equality `===`, but the difference is that `NaN` is considered equal to `NaN`. So `NaN` can be used as the key as well.

This algorithm can't be changed or customized.

Chaining

Every `map.set` call returns the map itself, so we can "chain" the calls:

```
map.set('1', 'str1')
  .set(1, 'num1')
  .set(true, 'bool1');
```

```
// array of [key, value] pairs
let map = new Map([
  ['1',  'str1'],
  [1,    'num1'],
  [true, 'bool1']
]);

alert( map.get('1') ); // str1
```

To make map from an obj use new Map(Objects.entries(obj))

Object.fromEntries: Object from Map

We've just seen how to create `Map` from a plain object with `Object.entries(obj)`.

There's `Object.fromEntries` method that does the reverse: given an array of `[key, value]` pairs, it creates an object from them:

```
let prices = Object.fromEntries([
  ['banana', 1],
  ['orange', 2],
  ['meat', 4]
]);

// now prices = { banana: 1, orange: 2, meat: 4 }

alert(prices.orange); // 2
```

We can use `Object.fromEntries` to get a plain object from `Map`.

E.g. we store the data in a `Map`, but we need to pass it to a 3rd-party code that expects a plain object.

Here we go:

```
let map = new Map();
map.set('banana', 1);
map.set('orange', 2);
map.set('meat', 4);

let obj = Object.fromEntries(map.entries()); // make a plain object (*)
```

Or we can also write obj = Object.fromEntries(map) as they're also iterable

Set

A `Set` is a special type collection – "set of values" (without keys), where each value may occur only once.

Its main methods are:

- `new Set([iterable])` – creates the set, and if an `iterable` object is provided (usually an array), copies values from it into the set.
- `set.add(value)` – adds a value, returns the set itself.
- `set.delete(value)` – removes the value, returns `true` if `value` existed at the moment of the call, otherwise `false`.
- `set.has(value)` – returns `true` if the value exists in the set, otherwise `false`.
- `set.clear()` – removes everything from the set.
- `set.size` – is the elements count.

Map

Map is a collection of keyed data items, just like an `Object`. But the main difference is that `Map` allows keys of any type.

Methods and properties are:

- `new Map()` – creates the map.
- `map.set(key, value)` – stores the value by the key.
- `map.get(key)` – returns the value by the key, `undefined` if `key` doesn't exist in map.
- `map.has(key)` – returns `true` if the `key` exists, `false` otherwise.
- `map.delete(key)` – removes the element (the key/value pair) by the key.
- `map.clear()` – removes everything from the map.
- `map.size` – returns the current element count.

Maps and Sets are ordered by their insertion order.

We can use Array.from(iterable) to create an array from an iterable item.

Object.fromEntries([[1,2]]) returns an obj

Object.entries({a:1, b:2}) returns an array of array Which is acceptable by map.

Object: keys, values, entries:

For instance:

```
let user = {
  name: "John",
  age: 30
};
```

- `Object.keys(user) = ["name", "age"]`
- `Object.values(user) = ["John", 30]`
- `Object.entries(user) = [ ["name","John"], ["age",30] ]`

Ignore elements using commas

Unwanted elements of the array can also be thrown away via an extra comma:

```
// second element is not needed
let [firstName, , title] = ["Julius", "Caesar", "Consul", "of the Roman
alert( title ); // Consul
```

In the code above, the second element of the array is skipped, the third one is assigned to `title`, and the rest of the array items are also skipped (as there are no variables for them).

Works with any iterable on the right-side

…Actually, we can use it with any iterable, not only arrays:

```
let [a, b, c] = "abc"; // ["a", "b", "c"]
let [one, two, three] = new Set([1, 2, 3]);
```

That works, because internally a destructuring assignment works by iterating over the right value. It's a kind of syntax sugar for calling `for..of` over the value to the right of `=` and assigning the values.

Looping with .entries()

In the previous chapter, we saw the Object.entries(obj) method.

We can use it with destructuring to loop over the keys-and-values of an object:

```
let user = {
  name: "John",
  age: 30
};

// loop over the keys-and-values
for (let [key, value] of Object.entries(user)) {
  alert(`${key}:${value}`); // name:John, then age:30
}
```

Swap variables trick

There's a well-known trick for swapping values of two variables using a destructuring assignment:

```
let guest = "Jane";
let admin = "Pete";

// Let's swap the values: make guest=Pete, admin=Jane
[guest, admin] = [admin, guest];

alert(`${guest} ${admin}`); // Pete Jane (successfully swapped!)
```

Here we create a temporary array of two variables and immediately destructure it in swapped order.

We can swap more than two variables this way.

Default values

If the array is shorter than the list of variables on the left, there will be no errors. Absent values are considered undefined:

```
let [firstName, surname] = [];

alert(firstName); // undefined
alert(surname); // undefined
```

If we want a "default" value to replace the missing one, we can provide it using `=`:

```
// default values
let [name = "Guest", surname = "Anonymous"] = ["Julius"];

alert(name);    // Julius (from array)
alert(surname); // Anonymous (default used)
```

Default values can be more complex expressions or even function calls. They are evaluated only if the value is not provided.

For instance, here we use the `prompt` function for two defaults:

```
// runs only prompt for surname
let [name = prompt('name?'), surname = prompt('surname?')] = ["Julius"];
```

In the examples above variables were declared right in the assignment: `let {…} = {…}`. Of course, we could use existing variables too, without `let`. But there's a catch.

This won't work:

```
let title, width, height;

// error in this line
{title, width, height} = {title: "Menu", width: 200, height: 100};
```

The problem is that JavaScript treats `{...}` in the main code flow (not inside another expression) as a code block. Such code blocks can be used to group statements, like this:

```
{
  // a code block
  let message = "Hello";
  // ...
  alert( message );
}
```

So here JavaScript assumes that we have a code block, that's why there's an error. We want destructuring instead.

To show JavaScript that it's not a code block, we can wrap the expression in parentheses `(...)`:

```
let title, width, height;

// okay now
({title, width, height} = {title: "Menu", width: 200, height: 100});
```

```
let options = {
  title: "My menu",
  items: ["Item1", "Item2"]
};

function showMenu({
  title = "Untitled",
  width: w = 100,  // width goes to w
  height: h = 200, // height goes to h
  items: [item1, item2] // items first element goes to item1, second to item2
}) {
  alert( `${title} ${w} ${h}` ); // My Menu 100 200
  alert( item1 ); // Item1
  alert( item2 ); // Item2
}

showMenu(options);
```

The full syntax is the same as for a destructuring assignment:

```
function({
  incomingProperty: varName = defaultValue
  ...
})
```

Please note that such destructuring assumes that `showMenu()` does have an argument. If we want all values by default, then we should specify an empty object:

```
showMenu({}); // ok, all values are default

showMenu(); // this would give an error
```

We can fix this by making `{}` the default value for the whole object of parameters:

```
function showMenu({ title = "Menu", width = 100, height = 200 } = {}) {
  alert( `${title} ${width} ${height}` );
}

showMenu(); // Menu 100 200
```

Let {a, b, c} = { a: 1, b:2, c: 3 } here in obj dedtructuring, the variable names on the right MUST be similar to property names on right, order does not matter.

If we want to change the name we can do, { a: newA, b: newB, c:newC } = {obj}

In case of obj, destructuring returns an obj instead of array.

Json:

In json there are NO backticks or double quotes and obj property names are double quoted.

Json supports all primitive data types, objs and arrays.

Function properties, symbolic keys and values, properties with value undefined.

Json supports nested objects but NOT circular refrences.

We can use a replacer function in the stringify to name the properties that must be replaced, it returns undefined for values to be skipped and replaced values for rest.

The object on which stringify is applied is wrapped in wrapper function.

Last argument is for the number of spaces in the indentation.

We can also use string instead of space.

We can use toJSON() method to define our own JSON.stringify.

```
let value = JSON.parse(str[, reviver]);
```

```
// title: (meetup title), date: (meetup date)
let str = '{"title":"Conference","date":"2017-11-30T12:00:00.000Z"}';
```

...And now we need to *deserialize* it, to turn back into JavaScript object.

Let's do it by calling `JSON.parse`:

```
let str = '{"title":"Conference","date":"2017-11-30T12:00:00.000Z"}';

let meetup = JSON.parse(str);

alert( meetup.date.getDate() ); // Error!
```

Whoops! An error!

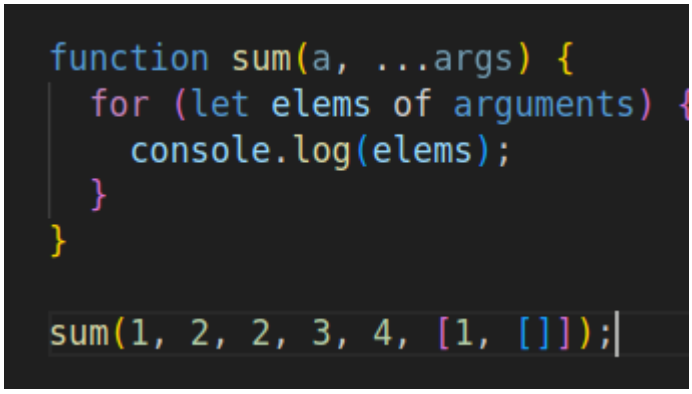
The value of `meetup.date` is a string, not a `Date` object. How could `JSON.parse` know that it should transform that string into a `Date`?

Let's pass to `JSON.parse` the reviving function as the second argument, that returns all values "as is", but `date` will become a `Date`:

```
let str = '{"title":"Conference","date":"2017-11-30T12:00:00.000Z"}';

let meetup = JSON.parse(str, function(key, value) {
  if (key == 'date') return new Date(value);
  return value;
});
```

Javascript has arguments property that stores ALL the arguments passed to the function.



```
function sum(a, ...args) {
  for (let elems of arguments) {
    console.log(elems);
  }
}

sum(1, 2, 2, 3, 4, [1, []]);
```

Dont use arguments as it IS array LIKE but not array so it does not support array methods.

Arrow function does not have it's own arguments so it takes it's parents

Easy way to cpy an array: newArr = [...arr]

Polyfills: add functions that are not available in the environment but are available in modern standard.

Try Catch Block:

`try...catch` only works for runtime errors

For `try...catch` to work, the code must be runnable. In other words, it should be valid JavaScript.

It won't work if the code is syntactically wrong, for instance it has unmatched curly braces:

```
try {
  {{{{{{{{{{{{
} catch (err) {
  alert("The engine can't understand this code, it's invalid");
}
```

The JavaScript engine first reads the code, and then runs it. The errors that occur on the reading phase are called "parse-time" errors and are unrecoverable (from inside that code). That's because the engine can't understand the code.

So, `try...catch` can only handle errors that occur in valid code. Such errors are called "runtime errors" or, sometimes, "exceptions".

`try...catch` works synchronously

If an exception happens in “scheduled” code, like in `setTimeout`, then `try...catch` won't catch it:

```
try {
  setTimeout(function() {
    noSuchVariable; // script will die here
  }, 1000);
} catch (err) {
  alert( "won't work" );
}
```

That's because the function itself is executed later, when the engine has already left the `try...catch` construct.

To catch an exception inside a scheduled function, `try...catch` must be inside that function:

```
setTimeout(function() {
  try {
    noSuchVariable; // try...catch handles the error!
  } catch {
    alert( "error is caught here!" );
  }
}, 1000);
```

`finally` and `return`

The `finally` clause works for *any* exit from `try...catch`. That includes an explicit `return`.

In the example below, there's a `return` in `try`. In this case, `finally` is executed just before the control returns to the outer code.

```
function func() {

  try {
    return 1;

  } catch (err) {
    /* ... */
  } finally {
    alert( 'finally' );
  }
}

alert( func() ); // first works alert from finally, and then this one
```

Let's imagine we've got a fatal error outside of `try...catch`, and the script died. Like a programming error or some other terrible thing.

Is there a way to react on such occurrences? We may want to log the error, show something to the user (normally they don't see error messages), etc.

There is none in the specification, but environments usually provide it, because it's really useful. For instance, Node.js has `process.on("uncaughtException")` for that. And in the browser we can assign a function to the special window.onerror property, that will run in case of an uncaught error.

Promises:

Callbacks creates Inversion of Control.

A `finally` handler doesn't get the outcome of the previous handler (it has no arguments). This outcome is passed through instead, to the next suitable handler.

If a `finally` handler returns something, it's ignored.

When `finally` throws an error, then the execution goes to the nearest error handler.

.then() always returns a promises so that we can call Another .then() on it.

```
class Thenable {
  constructor(num) {
    this.num = num;
  }
  then(resolve, reject) {
    alert(resolve); // function() { native code }
    // resolve with this.num*2 after the 1 second
    setTimeout(() => resolve(this.num * 2), 1000); // (**)
  }
}

new Promise(resolve => resolve(1))
  .then(result => {
    return new Thenable(result); // (*)
  })
  .then(alert); // shows 2 after 1000ms
```

.catch() at the end of the chain is similar to the try...catch

In a regular `try..catch` we can analyze the error and maybe rethrow it if it can't be handled. The same thing is possible for promises.

If we `throw` inside `.catch`, then the control goes to the next closest error handler. And if we handle the error and finish normally, then it continues to the next closest successful `.then` handler.

In the example below the `.catch` successfully handles the error:

```
// the execution: catch -> then
new Promise((resolve, reject) => {

  throw new Error("Whoops!");

}).catch(function(error) {

  alert("The error is handled, continue normally");

}).then(() => alert("Next successful handler runs"));
```

In the browser we can catch such errors using the event `unhandledrejection`:

```
window.addEventListener('unhandledrejection', function(event) {
  // the event object has two special properties:
  alert(event.promise); // [object Promise] - the promise that generated the
  alert(event.reason); // Error: Whoops! - the unhandled error object
});

new Promise(function() {
  throw new Error("Whoops!");
}); // no catch to handle the error
```

The event is the part of the HTML standard.

If an error occurs, and there's no `.catch`, the `unhandledrejection` handler triggers, and gets the `event` object with the information about the error, so we can do something.

Usually such errors are unrecoverable, so our best way out is to inform the user about the problem and probably report the incident to the server.

In non-browser environments like Node.js there are other ways to track unhandled errors.

Let's say we want many promises to execute in parallel and wait until all of them are ready.

For instance, download several URLs in parallel and process the content once they are all done.

That's what `Promise.all` is for.

The syntax is:

```
let promise = Promise.all(iterable);
```

`Promise.all` takes an iterable (usually, an array of promises) and returns a new promise.

The new promise resolves when all listed promises are resolved, and the array of their results becomes its result.

For instance, the `Promise.all` below settles after 3 seconds, and then its result is an array `[1, 2, 3]`:

```
Promise.all([
  new Promise(resolve => setTimeout(() => resolve(1), 3000)), // 1
  new Promise(resolve => setTimeout(() => resolve(2), 2000)), // 2
  new Promise(resolve => setTimeout(() => resolve(3), 1000))  // 3
]).then(alert); // 1,2,3 when promises are ready: each promise contributes an
```

If any of the promise is rejected, the promise returned by promise.all immediately rejects with that error, ignoring the further promises

`Promise.all(iterable)` allows non-promise "regular" values in `iterable`

Normally, `Promise.all(...)` accepts an iterable (in most cases an array) of promises. But if any of those objects is not a promise, it's passed to the resulting array "as is".

For instance, here the results are `[1, 2, 3]`:

```
Promise.all([
  new Promise((resolve, reject) => {
    setTimeout(() => resolve(1), 1000)
  }),
  2,
  3
]).then(alert); // 1, 2, 3
```

So we are able to pass ready values to `Promise.all` where convenient.

`Promise.allSettled` just waits for all promises to settle, regardless of the result. The resulting array has:

- `{status:"fulfilled", value:result}` for successful responses,
- `{status:"rejected", reason:error}` for errors.

For example, we'd like to fetch the information about multiple users. Even if one request fails, we're still interested in the others.

Let's use `Promise.allSettled`:

```
let urls = [
  'https://api.github.com/users/iliakan',
  'https://api.github.com/users/remy',
  'https://no-such-url'
];

Promise.allSettled(urls.map(url => fetch(url)))
  .then(results => { // (*)
    results.forEach((result, num) => {
      if (result.status == "fulfilled") {
        alert(`${urls[num]}: ${result.value.status}`);
      }
      if (result.status == "rejected") {
        alert(`${urls[num]}: ${result.reason}`);
      }
    });
  });
```

Promise.race(iterable) => gives the first settled(resolved or rejected) promise and ignores the rest.

Promise.any() => gives the first Fulfilled result. If all promises are rejected then the return promise is rejected with AggregateError

Promises are immutable. You cannot change it’s data

Promisification:

Conversion of a function that accepts a promise into a function that returns a promise.

Modern browsers allow top-level `await` in modules

In modern browsers, `await` on top level works just fine, when we’re inside a module. We’ll cover modules in article Modules, introduction.

For instance:

```
// we assume this code runs at top level, inside a module
let response = await fetch('/article/promise-chaining/user.json');
let user = await response.json();

console.log(user);
```

If we’re not using modules, or older browsers must be supported, there’s a universal recipe: wrapping into an anonymous async function.

Like this:

```
(async () => {
  let response = await fetch('/article/promise-chaining/user.json');
  let user = await response.json();
  ...
})();
```

Summary

The `async` keyword before a function has two effects:

1. Makes it always return a promise.
2. Allows `await` to be used in it.

The `await` keyword before a promise makes JavaScript wait until that promise settles, and then:

1. If it’s an error, an exception is generated — same as if `throw error` were called at that very place.
2. Otherwise, it returns the result.

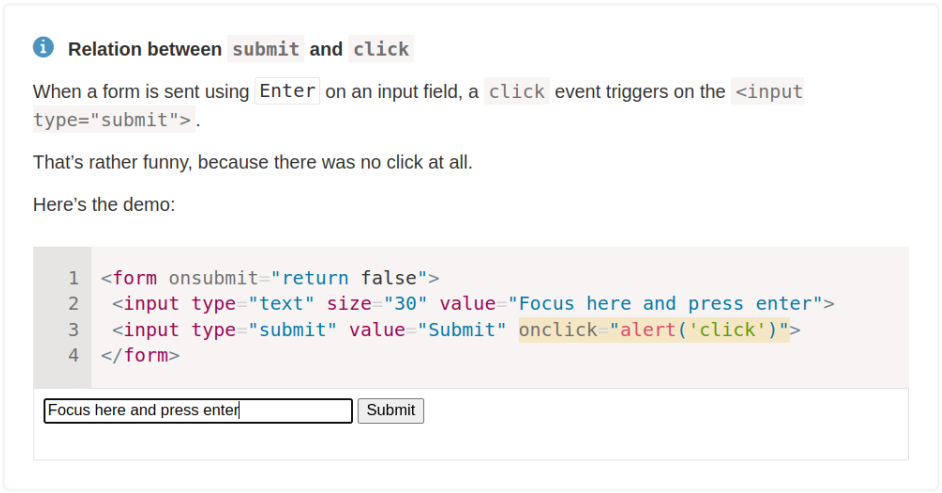
Together they provide a great framework to write asynchronous code that is easy to both read and write.

With `async/await` we rarely need to write `promise.then/catch`, but we still shouldn’t forget that they are based on promises, because sometimes (e.g. in the outermost scope) we have to use these methods. Also `Promise.all` is nice when we are waiting for many tasks simultaneously.

Await resolves the promise.

Async await is just a syntactical sugar for promise().then/catch.

We can ignore or override an error using catch, simply by returning a regular value.



Events:

Event handlers: a function that runs in case of an event

Html attribute names are not case sensitive

Don't use `setAttribute` for handlers.

Such a call won't work:

```
// a click on <body> will generate errors,
// because attributes are always strings, function becomes a string
document.body.setAttribute('onclick', function() { alert(1) });
```

DOM-property case matters.

Assign a handler to `elem.onclick`, not `elem.ONCLICK`, because DOM properties are case-sensitive.

Removal requires the same function

To remove a handler we should pass exactly the same function as was assigned.

This doesn't work:

```
elem.addEventListener( "click" , () => alert('Thanks!'));
// ....
elem.removeEventListener( "click", () => alert('Thanks!'));
```

The handler won't be removed, because `removeEventListener` gets another function – with the same code, but that doesn't matter, as it's a different function object.

Here's the right way:

```
function handler() {
  alert( 'Thanks!' );
}

input.addEventListener("click", handler);
// ....
input.removeEventListener("click", handler);
```

Please note – if we don't store the function in a variable, then we can't remove it. There's no way to "read back" handlers assigned by `addEventListener`.

By addEventListener we can add multiple event handlers to an event

`event.type`

Event type, here it's `"click"`.

`event.currentTarget`

Element that handled the event. That's exactly the same as `this`, unless the handler is an arrow function, or its `this` is bound to something else, then we can get the element from `event.currentTarget`.

`event.clientX` / `event.clientY`

Window-relative coordinates of the cursor, for pointer events.

ECMAScript:

JS

```
const person = "Mike";
const age = 28;

function myTag(strings, personExp, ageExp) {
  const str0 = strings[0]; // "That "
  const str1 = strings[1]; // " is a "
  const str2 = strings[2]; // "."

  const ageStr = ageExp < 100 ? "youngster" : "centenarian";

  // We can even return a string built using a template literal
  return `${str0}${personExp}${str1}${ageStr}${str2}`;
}

const output = myTag`That ${person} is a ${age}.`;

console.log(output);
// That Mike is a youngster.
```