# CT449:
# Phát triển ứng dụng web

Bùi Võ Quốc Bảo
(bvqbao@cit.ctu.edu.vn)

Cần Thơ, 2022

# Credit

- The slides are inspired by the CS193X course created by Victoria Kirst

# (Server-side) JavaScript

# Classes in JavaScript

# Public methods

```
class ClassName {
  constructor(params) {
    ...
  }
  methodName() {
    ...
  }
  methodName() {
    ...
  }
}
```

`constructor` is optional

Parameters for the constructor and methods are defined in the same they are for global functions

You do not use the `function` keyword to define methods

# Public methods

```
class ClassName {
  constructor(params) {
    ...
  }
  methodOne() {
    this.methodTwo();
  }
  methodTwo() {
    ...
  }
}
```

Within the class, you must always refer to other methods in the class with the **`this.`** prefix

# Public fields

```
class ClassName {
  fieldName;          // Optional
  constructor(params) {
    this.fieldName = fieldValue;
    this.fieldName = fieldValue;
  }
  methodName() {
    this.fieldName = fieldValue;
  }
}
```

Define public fields by setting **`this.`*fieldName*** in the constructor… or in any other function

# Public fields

```
class ClassName {
  constructor(params) {
    this.someField = someParam;
  }
  methodName() {
    const someValue = this.someField;
  }
}
```

Within the class, you must always refer to fields with the **`this.`** prefix

# Private fields/methods

```
class ClassName {
  #privateField     // Required

  constructor(params) {
    this.#privateField = fieldValue;
    this.publicField = fieldValue;
  }

  #privateMethodName() {
    this.#privateField = fieldValue;
  }
}
```

# Instantiation

Create new objects using the new keyword:

```
class SomeClass {
  ...
  someMethod() { … }
}

const x = new SomeClass();
const y = new SomeClass();
y.someMethod();
```

# First-class functions

# First-class functions

**Functions in JavaScript are objects**

- They can be saved in variables
- They can be passed as parameters
- They have properties, like other objects
- They can be defined without an identifier

(This is also called having **first-class functions**, i.e. functions in JavaScript are "first-class" because they are treated like any other variable/object)

# First-class functions

**Functions in JavaScript are objects**

- They can be saved in variables
- They can be passed as parameters
- They have properties, like other objects
- They can be defined without an identifier

(This is also called having **first-class functions**, i.e. functions in JavaScript are "first-class" because they are treated like any other variable/object)

???

# First-class functions

**Functions in JavaScript are objects**

- They can be saved in variables
- They can be passed as parameters
- They have properties, like other objects
- They can be defined without an identifier

(This is also called having **first-class functions**, i.e. functions in JavaScript are "first-class" because they are treated like any other variable/object)

???

Isn't there like... a fundamental difference between "code" and "data"?

# Back to the veeeeery basics

**What is code?**

- A list of instructions your computer can execute
- Each line of code is a **statement**

**What is a function?**

- A labeled group of **statements**
- The statements in a function are executed when the function is invoked

**What is a variable?**

- A labeled piece of **data**

# Objects in JS

Objects in JavaScript are sets of property-value pairs:

```
const bear = {
  name: 'Ice Bear',
  hobbies: ['knitting', 'cooking', 'dancing']
};
```

- Like any other value, Objects can be saved in **variables**
- Objects can be passed as parameters to functions

# Back to the veeeeery basics

**What is code?**

- A list of instructions your computer can execute
- Each line of code is a **statement**

**What is a function?**

- A labeled group of **statements**
- The statements in a function are executed when the function is invoked

**What is a variable?**

- A labeled piece of **data**

What could it mean for a function to be an object, i.e. a kind of data?

# Function variables

You can declare a function in several ways:

```
function myFunction(params) {
}

const myFunction = function(params) {
}

const myFunction = (params) => {
}
```

# Function variables

```
function myFunction(params) {
}

const myFunction = function(params) {
}

const myFunction = (params) => {
}
```

Functions are invoked in the same way, regardless of how they were declared:

```
myFunction();
```

```
const x = 15;
let y = true;

const greeting = function() {
  console.log('hello, world');
}
```

**"A function in JavaScript is an object of type `Function`"**

**In the interpreter's memory:**

```
const x = 15;
let y = true;

const greeting = function() {
  console.log('hello, world');
}
```

**"A function in JavaScript is an object of type `Function`"**

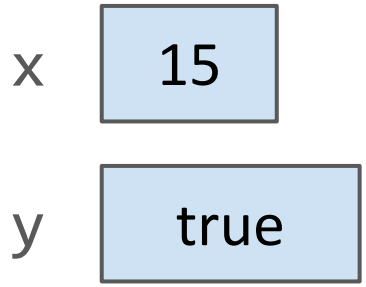
**In the interpreter's memory:**


```
const x = 15;
let y = true;

const greeting = function() {
  console.log('hello, world');
}
```

**"A function in JavaScript is an object of type `Function`"**

**In the interpreter's memory:**

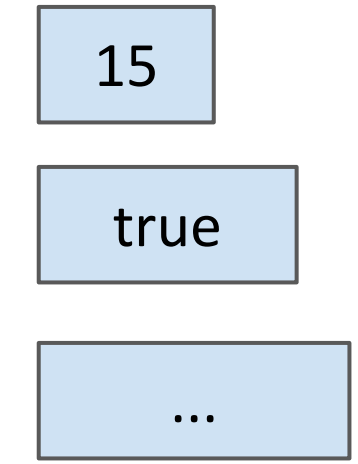



```
const x = 15;
let y = true;

const greeting = function() {
  console.log('hello, world');
}
```

**"A function in JavaScript is an object of type `Function`"**

**In the interpreter's memory:**




```
const x = 15;
let y = true;

const greeting = function() {
  console.log('hello, world');
}
```

**"A function in JavaScript is an object of type `Function`"**

What this really means:

- When you declare a function, there is an object of type `Function` that gets created alongside the labeled block of executable code

# Function properties

```
const greeting = function() {
  console.log('hello, world');
}

console.log(greeting.name);
console.log(greeting.toString());
```

When you declare a function, you create an object of type `Function`, which has properties like:

- `name`
- `toString`

# `Function` properties

```
const greeting = function() {
  console.log('hello, world');
}

greeting.call();
```

`Function` objects also have a `call` method, which invokes the underlying executable code associated with this function object

# `Function` properties

```
const greeting = function() {
  console.log('hello, world');
}

greeting.call();
greeting();
```

`()` is an operation on the `Function` object (spec)

- When you use the `()` operator on a Function object, it is calling the object's `call()` method, which in turn executes the function's underlying code

# Code vs `Functions`

**Important distinction:**

- Function, the executable code
  - A group of instructions to the computer
- `Function`, the object
  - A JavaScript object, i.e. a set of property-value pairs
  - `Function` objects have executable code associated with them
  - This executable code can be invoked by
    - *functionName*`();` *or*
    - *functionName*`.call();`

# **Note:** `Function` is special

Only `Function` objects have executable code associated with them

- Regular JS objects **cannot** be invoked
- Regular JS objects **cannot** be given executable code
  - I.e. you can't make a regular JS object into a callable function

```
const bear = {
  name: 'Ice Bear',
  hobbies: ['knitting', 'cooking', 'dancing']
};
bear(); // error!
```

# Function Objects vs Objects

```
function sayHello() {
  console.log('Ice Bear says hello');
}

const bear = {
  name: 'Ice Bear',
  hobbies: ['knitting', 'cooking', 'dancing'],
  greeting: sayHello
};
bear.greeting();
```

But you can give your object `Function` properties and then invoke those properties

# Function Objects vs Objects

```
function sayHello() {
  console.log('Ice Bear says hello');
}

const bear = {
  name: 'Ice Bear',
  hobbies: ['knitting', 'cooking', 'dancing'],
  greeting: sayHello
};
bear.greeting();
```

The `greeting` property is an object of `Function` type

# Callbacks

**Callback**: A function that's passed as a parameter to another function, usually in response to something

```
app.get('/hello', function (req, res) {
  res.send('GET hello!');
});
```

Because every function declaration creates a `Function` object, we can pass `Functions` as parameters to other functions

# findIndex

```
const flavors =
    ['vanilla', 'chocolate', 'strawberry', 'green tea'];


function isStrawberry(element) {
  return element === 'strawberry';
}

const indexOfStrawberry = flavors.findIndex(isStrawberry);
```

The **isStrawberry** function will fire for each element in the array

# findIndex

```
const flavors =
    ['vanilla', 'chocolate', 'strawberry', 'green tea'];


function isStrawberry(element) {
  return element === 'strawberry';
}

const indexOfStrawberry = flavors.findIndex(isStrawberry);
```

# findIndex

```
const flavors =
    ['vanilla', 'chocolate', 'strawberry', 'green tea'];
```

```
function isStrawberry(element) {
  return element === 'strawberry';
}
```

Returns **false**, so keep searching.

```
const indexOfStrawberry = flavors.findIndex(isStrawberry);
```

# findIndex

```
const flavors =
    ['vanilla', 'chocolate', 'strawberry', 'green tea'];


function isStrawberry(element) {
  return element === 'strawberry';
}

const indexOfStrawberry = flavors.findIndex(isStrawberry);
```

# findIndex

```
const flavors =
    ['vanilla', 'chocolate', 'strawberry', 'green tea'];
```

```
function isStrawberry(element) {
  return element === 'strawberry';
}
```

Returns **false**, so keep searching.

```
const indexOfStrawberry = flavors.findIndex(isStrawberry);
```

# findIndex

```
const flavors =
    ['vanilla', 'chocolate', 'strawberry', 'green tea'];


function isStrawberry(element) {
  return element === 'strawberry';
}

const indexOfStrawberry = flavors.findIndex(isStrawberry);
```

# findIndex

```
const flavors =
    ['vanilla', 'chocolate', 'strawberry', 'green tea'];
```

```
function isStrawberry(element) {
  return element === 'strawberry';
}
```

Returns **true**, so stop searching.

```
const indexOfStrawberry = flavors.findIndex(isStrawberry);
```

# findIndex

```
const flavors =
    ['vanilla', 'chocolate', 'strawberry', 'green tea'];
```

```
function isStrawberry(element) {
  return element === 'strawberry';
}

const indexOfStrawberry = flavors.findIndex(isStrawberry);
```

findIndex returns 2, since the first element to pass the **testing function** was found at index 2

# Anonymous functions

# Anonymous functions

We do not need to give an identifier to functions

When we define a function without an identifier, we call it an **anonymous function**

- Also known as a **function literal**, or a **lambda function**

**We can define our test function directly in `findIndex`:**

```
function isStrawberry(element) {
  return element === 'strawberry';
}

const index = flavors.findIndex(isStrawberry);
```

# Anonymous functions

We do not need to give an identifier to functions.

When we define a function without an identifier, we call it an **anonymous function**

- Also known as a **function literal**, or a **lambda function**

**We can define our test function directly in `findIndex`:**

```
const index = flavors.findIndex(
  function(element) { return element === 'strawberry'; });
```

# Arrow functions

We can use the **arrow function** syntax for defining functions:

```
const index = flavors.findIndex(
  function(element) { return element === 'strawberry'; });
```

# Arrow functions

We can use the **arrow function** syntax for defining functions:

```
const index = flavors.findIndex(
 (element) => { return element === 'strawberry'; });
```

# Concise arrow functions

We can use the **concise version** of the arrow function:

- You can omit the parentheses if there is only one parameter
- You can omit the curly braces if there's only one statement in the function, and it's a return statement

```
const index = flavors.findIndex(
 (element) => { return element === 'strawberry'; });
```

# Concise arrow functions

We can use the **concise version** of the arrow function:

- You can omit the parentheses if there is only one parameter
- You can omit the curly braces if there's only one statement in the function, and it's a return statement

```
const index = flavors.findIndex(
   element => element === 'strawberry');
```

# Case-insensitive search

If we wanted to make this case insensitive, we could do:

```
const index = flavors.findIndex(
   element => element.toLowerCase() === 'strawberry');
```

# Case-insensitive search

If we wanted to make this case insensitive, we could do:

```
const index = flavors.findIndex(
   element => element.toLowerCase() === 'strawberry');
```

This is a lot more elegant than the for-loop approach!

```
for (let i = 0; i < flavors.length; i++) {
  if (flavors[i].toLowerCase() === 'strawberry') {
    break;
  }
}
const index = i;
```

# map

E.g., Map an array of objects to an array of strings:

```
const persons = [
  {firstname : "Malcom", lastname: "Reynolds"},
  {firstname : "Kaylee", lastname: "Frye"},
  {firstname : "Jayne", lastname: "Cobb"}
];

console.log(persons.map((p) => {
  return [p.firstname, p.lastname].join(" ");
}));
```

# reduce

E.g., Find sum of elements in an array:

```
const numbers = [ 1, 2, 5, 3, 6, 7 ];
const sum = numbers.reduce(
      (acc, num) => acc + num,
      0    // Init value for acc
);

console.log(sum);
```

# Arrow functions and **this** keyword

An arrow function doesn't have its own bindings to **this** or **super**, and **should not** be used as methods

```
const obj = {
      i: 10,
      b: () => console.log(this.i, this),
      c: function() {
            console.log(this.i, this);
      }
};
obj.b();   // undefined {}
obj.c();   // 10 { i: 10, b: [Function: b], … }
```

# Currying

# isFlavor

What if instead of checking specifically for `strawberry`...

```
function isStrawberry(element) {
  return element === 'strawberry';
}
```

# isFlavor

...we wanted to create a generic `isFlavor` checker?

```
function isFlavor(flavor, element) {
  return element === flavor;
}
```

# isFlavor

```
const flavors =
    ['vanilla', 'chocolate', 'strawberry', 'green tea'];


function isFlavor(element) {
  // ERROR: flavor is undefined!
  return element === flavor;
}

const indexOfFlavor = flavors.findIndex(isFlavor);
```

The problem is there's no way to pass in the `flavor` parameter in the callback for `findIndex`...

# Currying

```
const flavors =
    ['vanilla', 'chocolate', 'strawberry', 'green tea'];

function createFlavorTest(flavor) {
  function isFlavor(element) {
    return element === flavor;
  }
  return isFlavor;
}
const isStrawberry = createFlavorTest('strawberry');
const indexOfFlavor = flavors.findIndex(isStrawberry);
```

**Solution:** Create a function that takes a `flavor` parameter and creates a testing function for that parameter

# *Aside*: closure

```
const flavors =
    ['vanilla', 'chocolate', 'strawberry', 'green tea'];

function createFlavorTest(flavor) {
  function isFlavor(element) {
    return element === flavor;
  }
  return isFlavor;
}
const isStrawberry = createFlavorTest('strawberry');
const indexOfFlavor = flavors.findIndex(isStrawberry);
```

*Aside*: Any function that is declared within another function is called a **closure (??? Revise)**. Closures can refer to variables in the outer function (`flavor` in this case)

# Currying

```
function isFlavor(flavor, element) {
  return element === flavor;
}
```

```
function createFlavorTest(flavor) {
  function isFlavor(element) {
    return element === flavor;
  }
  return isFlavor;
}
```

```
flavors.findIndex(isFlavor);
```

This idea is called **currying**: breaking down a function with multiple arguments by applying one at a time in a sequence of created functions

func(a, b, c) => func( a )( b )( c )

# Review: Functional JavaScript

Functions in JavaScript are **first-class citizens**:

- Objects that can be passed as parameters
- Can be created within functions:
  - Inner functions are called **closures**
- Can be created without being saved to a variable
  - These are called **anonymous functions**, or function literals, or lambdas
- Can be created and returned from functions
  - Constructing a new function that references part of the outer function's parameters is called **currying**

# `Promises`:
# Another conceptual odyssey

# `Promises` and `.then()`

A `Promise`:

- An object representing the eventually result of an asynchronous operation
- Has a `then()` method that lets you attach functions to execute onSuccess or onError
- Allows you to build **chains** of asynchronous results

Promises are one way to deal with asynchronous code, without getting stuck in callback hell.

# Fetch API

```
npm install node-fetch@2
```

Không cần thiết nếu dùng node v18.x

```
const fetch = require('node-fetch');

const url = 'https://jsonplaceholder.typicode.com/users';

function onSuccess(response) { … }
function onFail(error) { … }
fetch(url).then(onSuccess, onFail);
```

# Promise syntax

**Q: How does this syntax work?**

```
fetch(url).then(onSuccess, onFail);
```

# Promise syntax

**Q: How does this syntax work?**

```
fetch(url).then(onSuccess, onFail);
```

**The syntax above is the same as:**

```
const promise = fetch(url);
promise.then(onSuccess, onFail);
```

# Promise syntax

```
const promise = fetch(url);
promise.then(onSuccess, onFail);
```

The object `fetch` returns is of type `Promise`

A promise is in one of three states:

- **pending**: initial state, not fulfilled or rejected
- **fulfilled**: the operation completed successfully
- **rejected**: the operation failed

You attach handlers to the promise via `.then()`

# Promise chaining

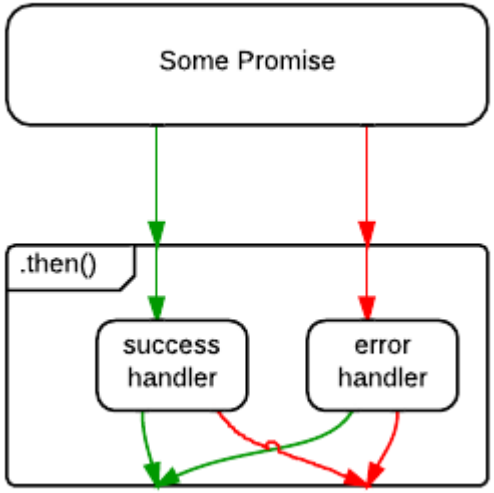
.then() returns a promise, so we can call a next .then() on it

```
new Promise((resolve, reject) => {
      setTimeout(() => resolve(1), 1000);
}).then(result => result * 2)
  .then(result => result * 2)
  .then((result) => {
      console.log(result); // 4
});
```

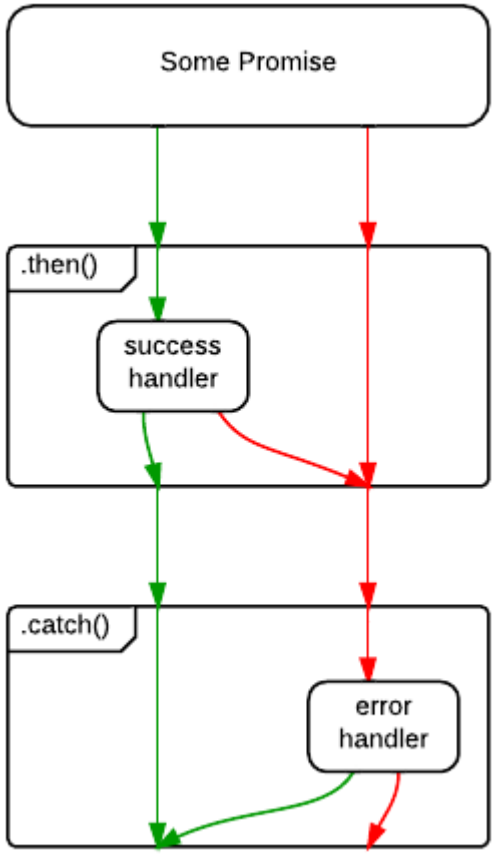
# Error handling

In practice, .catch() is usually used with .then()

```
fetch(url)
      .then(onSuccess)
      .catch(onFail);
```




.then()




.then().catch()

# Promise.all(iterable)

`Promise.all()`:

- Returns a single Promise that resolves to an array of the results of the input promises
- It rejects immediately upon **any** of the input promises rejecting

```
const promise1 = Promise.resolve(3);
const promise2 = 42;
const promise3 = new Promise((resolve, reject) => {
  setTimeout(resolve, 100, 'foo');
});

Promise.all([promise1, promise2, promise3]).then((values) => {
  console.log(values);
});
// expected output: Array [3, 42, "foo"]
```

# Promise.allSettled(iterable)

`Promise.allSettled()`: wait for all input promises to complete, regardless of whether or not one rejects

```
const promise1 = Promise.resolve(3);
const promise2 = new Promise(
  (resolve, reject) => setTimeout(reject, 100, 'foo'));

Promise.allSettled([promise1, promise2]).
  then((results) => results.forEach(
    (result) => console.log(result.status)));

// expected output:
// "fulfilled"
// "rejected"
```

# async/await

# Asynchronous `fetch()`

The usual asynchronous `fetch()` looks like this:

```
function onJsonReady(json) {
  console.log(json);
}

function onResponse(response) {
  return response.json();
}

fetch(url)
    .then(onResponse)
    .then(onJsonReady);
```

# Synchronous `fetch()`?

A hypothetical synchronous `fetch()` might look like this:

```
// THIS CODE DOESN'T WORK
const response = fetch(url);
const json = response.json();
console.log(json);
```

**This is a lot cleaner code-wise!!**

**However,** a synchronous `fetch()` would freeze the event loop as the resource was downloading, which would be terrible for performance

# async / await

What if we could get the best of both worlds?

- Synchronous-*looking* code
- That actually ran asynchronously

```
// THIS CODE DOESN'T WORK
const response = fetch(url);
const json = response.json();
console.log(json);
```

# async / await

What if we could get the best of both worlds?

- Synchronous-*looking* code
- That actually ran asynchronously

```
// But this code does work:
async function loadJson(url) {
  const response = await fetch(url);
  const json = await response.json();
  console.log(json);
}
loadJson('https://jsonplaceholder.typicode.com/users');
```

# async / await

What if we could get the best of both worlds?

- Synchronous-*looking* code
- That actually ran asynchronously

```
// But this code does work:
async function loadJson(url) {
  const response = await fetch(url);
  const json = await response.json();
  console.log(json);
}
loadJson('https://jsonplaceholder.typicode.com/users');
```

???

# `async` functions

A function marked `async` has the following qualities:

- It will behave more or less like a normal function if you don't put `await` expression in it

- An `await` expression is of form:
  - `await` ***promise***

# `async` functions

A function marked `async` has the following qualities:

- If there is an `await` expression, **the execution of the function will pause** until the `Promise` in the `await` expression is resolved
  - Note: The event loop is not blocked; it will continue processing other events as the async function is paused
- Then when the `Promise` is resolved, the execution of the function continues
- The `await` expression evaluates to the resolved value of the `Promise`

```
function onJsonReady(json) {
  console.log(json);
}
function onResponse(response) {
  return response.json();
}
fetch(url)
    .then(onResponse)
    .then(onJsonReady);
```

The methods in purple return Promises

---

```
async function loadJson(url) {
  const response = await fetch(url);
  const json = await response.json();
  console.log(json);
}
loadJson('https://…');
```

```
function onJsonReady(json) {
  console.log(json);
}
function onResponse(response) {
  return response.json();
}
fetch(url)
    .then(onResponse)
    .then(onJsonReady);
```

The variables in blue are the values that the `Promises` "resolve to"

---

```
async function loadJson(url) {
  const response = await fetch(url);
  const json = await response.json();
  console.log(json);
}
loadJson('https://…');
```

# async functions

```
async function loadJson(url) {
  const response = await fetch(url);
  const json = await response.json();
  console.log(json);
}
loadJson('https://…');
```

# async functions

```
async function loadJson(url) {
  const response = await fetch(url);
  const json = await response.json();
  console.log(json);
}
loadJson('https://…');
```

# async functions

```
async function loadJson(url) {
  const response = await fetch(url);
  const json = await response.json();
  console.log(json);
}
loadJson('https://…');
```

Since we've reached an `await` statement, two things happen:

1. `fetch(url);` runs
2. The execution of the `loadJson` function is paused here until `fetch(url);` has completed

# async functions

```
async function loadJson(url) {
  const response = await fetch(url);
  const json = await response.json();
  console.log(json);
}
loadJson('https://…');
console.log('after loadJson');
```

At the point, the JavaScript engine will return from `loadJson()` and it will continue executing where it left off

# async functions

```
async function loadJson(url) {
  const response = await fetch(url);
  const json = await response.json();
  console.log(json);
}
loadJson('https://…');
console.log('after loadJson');
```

# async functions

```
async function loadJson(url) {
  const response = await fetch(url);
  const json = await response.json();
  console.log(json);
}
loadJson('https://…');
console.log('after loadJson');
```

# async functions

```
async function loadJson(url) {
  const response = await fetch(url);
  const json = await response.json();
  console.log(json);
}
loadJson('https://…');
console.log('after loadJson');
```

# async functions

```
async function loadJson(url) {
  const response = await fetch(url);
  const json = await response.json();
  console.log(json);
}
loadJson('https://…');
console.log('after loadJson');
```

If there are other events and we had a event handler for it, JavaScript will continue executing those events

# async functions

```
async function loadJson(url) {
  const response = await fetch(url);
  const json = await response.json();
  console.log(json);
}
loadJson('https://…');
console.log('after loadJson');
```

When the `fetch()` completes, the JavaScript engine will resume execution of `loadJson()`

# Recall: `fetch()` resolution

```
function onResponse(response) {
  return response.json();
}
fetch(url)
    .then(onResponse);
```

Normally when `fetch()` finishes, it executes the `onResponse` callback, whose parameter will be response

**In `Promise`-speak:**

- The return value of `fetch()` is a `Promise` that **resolves to** the `response` object

# async functions

```
async function loadJson(url) {
  const response = await fetch(url);
  const json = await response.json();
  console.log(json);
}
loadJson('https://…');
console.log('after loadJson');
```

The value of the `await` expression is the value that the `Promise` resolves to, in this case `response`

# async functions

```
async function loadJson(url) {
  const response = await fetch(url);
  const json = await response.json();
  console.log(json);
}
loadJson('https://…');
console.log('after loadJson');
```

# async functions

```
async function loadJson(url) {
  const response = await fetch(url);
  const json = await response.json();
  console.log(json);
}
loadJson('https://…');
```

Since we've reached an `await` statement, two things happen:

1. `response.json();` runs
2. The execution of the `loadJson` function is paused here until `response.json();` has completed

# async functions

```
async function loadJson(url) {
  const response = await fetch(url);
  const json = await response.json();
  console.log(json);
}
loadJson('https://…');
```

If there are other events and we had a event handler for it, JavaScript will continue executing those events

# async functions

```
async function loadJson(url) {
  const response = await fetch(url);
  const json = await response.json();
  console.log(json);
}
loadJson('https://…');
```

When the `response.json()` completes, the JavaScript engine will resume execution of `loadJson()`

# Recall: `json()` resolution

```
function onJsonReady(jsObj) {
  console.log(jsObj);
}
function onResponse(response) {
  return response.json();
}
fetch(url)
    .then(onResponse)
    .then(onJsonReady);
```

Normally when `json()` finishes, it executes the `onJsonReady` callback, whose parameter will be **`jsObj`**

**In `Promise`-speak:**

- The return value of `json()` is a `Promise` that **resolves to** the **`jsObj`** object

# async functions

```
async function loadJson(url) {
  const response = await fetch(url);
  const json = await response.json();
  console.log(json);
}
loadJson('https://…');
```

The value of the `await` expression is the value that the `Promise` resolves to, in this case `json`

# async functions

```
async function loadJson(url) {
  const response = await fetch(url);
  const json = await response.json();
  console.log(json);
}
loadJson('https://…');
```

# async functions

```
async function loadJson(url) {
  const response = await fetch(url);
  const json = await response.json();
  console.log(json);
}
loadJson('https://…');
```

# async functions

```
async function loadJson(url) {
  const response = await fetch(url);
  const json = await response.json();
  console.log(json);
}
loadJson('https://…');
```

Note that the JS execution does *not* return back to the call site, since the JS execution already did that when we saw the first `await` expression

# Returning from async

**Q: What happens if we return a value from an async function?**

```
async function loadJson(url) {
  const response = await fetch(url);
  const json = await response.json();
  console.log(json);
  return true;
}
loadJson('https://…');
```

# Returning from async

**A: async functions must always return a `Promise`**

```
async function loadJson(url) {
  const response = await fetch(url);
  const json = await response.json();
  console.log(json);
  return true;
}
loadJson('https://…');
```

If you return a value that is **not** a `Promise` (such as `true`), then the JavaScript engine will automatically wrap the value in a `Promise` that resolves to the value you returned

# Returning from async

```
function loadJsonDone(value) {
  console.log('loadJson complete!');
  console.log('value: ' + value); // value: true
}

async function loadJson(url) {
  const response = await fetch(url);
  const json = await response.json();
  console.log(json);
  return true;
}
loadJson('https://…').then(loadJsonDone);
console.log('after loadJson');
```

# Error handling with async/await

```
async function loadJson(url) {
  try {
      const response = await fetch(url);
      const json = await response.json();
      console.log(json);
   } catch (error) {
      console.log(error.message);
   } finally {
      console.log('Done');
   }
}
loadJson('https://…');
```

# Error handling with async/await

```
const handlePromise = promise => {
    return promise.then(data => [null, data])
       .catch(error => [error, undefined]);
}

async function fetchJson() {
  let response = await fetch('http://…');
  if (!response.ok) {
    throw new Error(`Error: ${response.status}`);
  }
  return response.json();
}

let [error, json] = await handlePromise(fetchJson());
```

# More async

- Constructors cannot be marked async
  - A constructor returns the object being created while an async method returns a promise

- But you can pass async functions as parameters to wherever you can pass a function as a parameter

# To get the return value from a function in JavaScript ???

It depends on the function in question (check its docs)

```
// funct is synchronous
const retValue = funct();

// funct is asynchronous, callback version
funct(function(retValue) { … });

// funct is asynchronous, Promise version
funct().then(function(retValue) { … });

(async function caller() {
      const retValue = await funct();
})();
```

# JSON

# JavaScript Object Notation

**JSON**: stands for **J**ava**S**cript **O**bject **N**otation

- Created by Douglas Crockford
- Defines a way of **serializing** JavaScript objects
  - **to serialize**: to turn an object into a string that can be deserialized
  - **to deserialize**: to turn a serialized string into an object
- Built on two structures: a collection of name/value pairs and an ordered list of values

# JavaScript Object Notation

**JSON**: stands for **J**ava**S**cript **O**bject **N**otation

- A value can be a *string in double quotes*, or a *number*, or *true* or *false* or *null*, or *an object* or *an array*. These structures can be nested
- Don't support comments

- `JSON.stringify(`**object**`)` returns a string representing **object** serialized in JSON format
- `JSON.parse(`**jsonString**`)` returns a JS object from the **jsonString** serialized in JSON format

# JSON.stringify()

We can use the `JSON.stringify()` function to seralize a JavaScript object:

```
const bear = {
  name: 'Ice Bear',
  hobbies: ['knitting', 'cooking', 'dancing']
};

const serializedBear = JSON.stringify(bear);
console.log(serializedBear);
```

# JSON.parse()

We can use the `JSON.parse()` function to deseralize a JavaScript object:

```
const bearString = '{
  "name":"Ice Bear",
  "hobbies":["knitting","cooking","dancing"]
}';

const bear = JSON.parse(bearString);
console.log(bear);
```

# Why JSON?

JSON is a useful format for storing data that we can load into a JavaScript API

Let's say we had a list of Songs and Titles

- If we stored it as a text file, we would have to know how we are separating song name vs title, etc
- If we stored it as a JSON file, we can just deserialize the object

# JSON

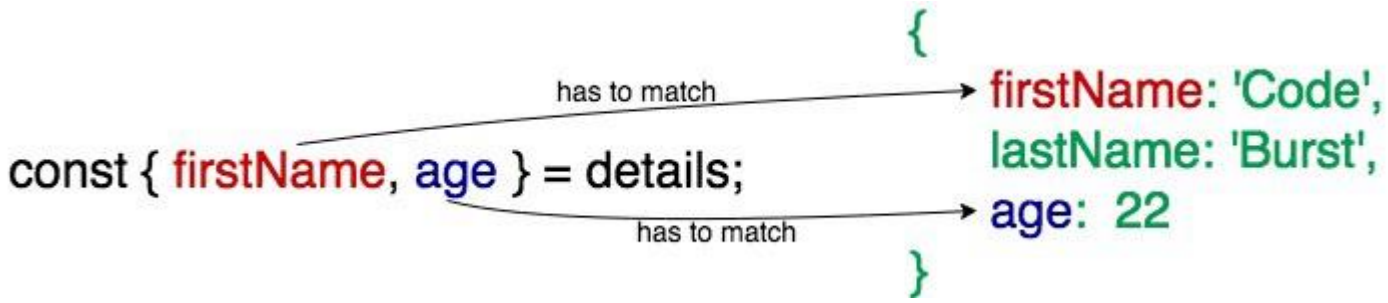

```
[
  {
    "id": 1,
    "name": "Leanne Graham",
    "username": "Bret",
    "email": "Sincere@april.biz",
    "address": {
      "street": "Kulas Light",
      "suite": "Apt. 556",
      "city": "Gwenborough",
      "zipcode": "92998-3874",
      "geo": {
        "lat": "-37.3159",
        "lng": "81.1496"
      }
    },
    "phone": "1-770-736-8031 x56442",
    "website": "hildegard.org",
    "company": {
      "name": "Romaguera-Crona",
      "catchPhrase": "Multi-layered client-server neural-net",
      "bs": "harness real-time e-markets"
    }
  },
  {
    "id": 2,
    "name": "Ervin Howell",
    "username": "Antonette",
    "email": "Shanna@melissa.tv",
    "address": {
      "street": "Victor Plains",
      "suite": "Suite 879",
      "city": "Wisokyburgh",
      "zipcode": "90566-7771",
      "geo": {
        "lat": "-43.9509",
        "lng": "-34.4618"
      }
    },
    "phone": "010-692-6593 x09125",
    "website": "anastasia.net",
    "company": {
      "name": "Deckow-Crist",
      "catchPhrase": "Proactive didactic contingency",
      "bs": "synergize scalable supply-chains"
    }
  },
```

# Some other features/syntax

# Destructuring arrays/objects

```
let [a, b] = [ 1, 2, 3, 4 ];
console.log(a); // 1
console.log(b); // 2

let details = { firstName: 'Code', lastName:
'Burst', age: 22 };
let { firstName, age } = details;
```



# Alias variables

```
const { identifier: alias } = expression;

const hero = {
  name: 'Batman'
};
// Object destructuring:
const { name: heroName } = hero;
console.log(heroName); // 'Batman'
```

# Dynamic property names

```
const x = 'name';
const a = { [x]: 'Batman' }
console.log(a.name);  // 'Batman'
console.log(a[x]);    // 'Batman'

const property = 'name';
const hero = {
  name: 'Batman'
};
const { [property]: heroName } = hero;
Console.log(heroName); // 'Batman'
```

# Spread operator (...)

Operator (...) allows us to quickly copy all or part of an existing array or object into another array or object

```
let numberStore = [0, 1, 2];
let newNumber = 12;
numberStore = [...numberStore, newNumber];
=> numberStore = [0, 1, 2, 12]

let arr = [1, 2, 3];
let arr2 = [...arr];
=> arr2 = [1, 2, 3]
```

# Spread operator (...)

Operator (...) allows us to quickly copy all or part of an existing array or object into another array or object

```
let arr1 = [0, 1, 2];
let arr2 = [3, 4, 5];
arr1 = [...arr1, ...arr2];
⇒arr1 = [0, 1, 2, 3, 4, 5]

const numbers = [1, 2, 3, 4, 5, 6];
const [one, two, ...rest] = numbers;
=> [one, two, ...rest] = [0, 1, 2, 3, 4, 5]
```

# Spread operator (...)

```
let obj1 = { foo: 'bar', x: 42 };
let obj2 = { foo: 'baz', y: 13 };

let clonedObj = { ...obj1 };
// Object { foo: "bar", x: 42 }

let mergedObj = { ...obj1, ...obj2 };
// Object { foo: "baz", x: 42, y: 13 }
```

Notice the properties that did not match were combined, but the property that did match, color, was overwritten by the last object that was passed, `mergedObj`. The resulting foo is now ‘baz’.

# Rest operator (...)

```
const numbers = [1, 2, 3];
const [ first, ...restOfTheNumbers ] = numbers;

const [ first, ...restOfTheLetters ] = 'webdev';

const details = {
    firstName: 'Code',
    lastName: 'Burst',
    age: 22
};
const { age, ...restOfTheDetails } = details;
```

# Module Systems

# The need for modules

- Having a way to split the codebase into multiple files
- Allowing code resuse across different projects
- Encapsulation/Information hiding
- Managing dependencies

The distinction between *a module* and *a module system*

- A module: an actual unit of software (i.e., a .js file)
- A module system: syntax and tooling that allows us to define and use modules

# The need for modules

JavaScript had been lacking this feature for a long time

- Splitting the codebase into multiple files and importing them by using different <script> tags was good enough
- Immediately Invoked Function Expression (IIFE) pattern is used to create a private scope, exporting only public parts

# Immediately Invoked Function Expression (IIFE)

```
const myModule = (() => {
      const privateFoo = () => {};
      const privateBar = [];
      const exported = {
            publicFoo: () => {};
            publicBar: () => {};
      }
      return exported;
})();

myModule.publicFoo();
```

# CommonJS modules

- Each file is treated as a separate module
- The module.exports is a special object which is included in every JavaScript file in the Node.js application by default
- Whatever you assign to module.exports can be exposed to other modules/files
  - `require(path/to/file)`
  - `require(path/to/folder):` index.js file in the folder will be used (for nodejs)

# CommonJS modules

```
// circle.js
const { PI } = Math;
exports.area = (r) => PI * r ** 2;
exports.circumference = (r) => 2 * PI * r;

// app.js
const circle = require('./circle.js');
console.log(`The area of a circle of radius 4 is
${circle.area(4)}`);
```

# CommonJS modules

```
// square.js
class Square {
  constructor(width) { this.width = width; }
  area() { return this.width ** 2; }
};
module.exports = Square;

// app.js
const Square = require('./square.js');
const mySquare = new Square(2);
console.log(`The area of mySquare is
${mySquare.area()}`);
```

# ECMAScript modules

- Official standard format to package JS code for reuse
- Use import/export instead of module.exports/require


- Full support from Node.js 13.2.0 (also supported in most browsers)
- *For Node.js*, must use .mjs file extension, or put "type": "module" in the nearest *package.json* file

# ECMAScript modules

```
// circle.mjs
const { PI } = Math;
export const area = (r) => PI * r ** 2;
export const circumference = (r) => 2 * PI * r;
export default const baseCircle = {
      r: 10,
      printInfo() {
            console.log(this.r, area(this.r),
                  circumference(this.r));
      }
};
```

**Named export**

**Default export**

Note: Only **one** default export per module

# ECMAScript modules

```
// app.mjs
// import named exports within curly braces with
      // the same name
import circle, { area, circumference as c }
            from './circle.mjs';

// a default export can be imported with any name
import stdCircle from './circle.mjs';

// or import everything
import * as circle from './circle.mjs';
// circle.default to access the default export (if any)
```

# Module bundling

- The process of stitching together a group of modules (and their dependencies) into a single file (or group of files) in the correct order*
  - Usually involve some optimizations (i.e., removing spaces to reduce file size)

- Write code using a certain module system/syntax (i.e., es6) and convert to different module system/syntax (i.e., cjs, iife)

- Several tools available: *webpack, rollup, parcel*,…

*https://www.freecodecamp.org/news/javascript-modules-part-2-module-bundling-5020383cf306/