JavaScript Essentials

In this presentation

- JS
  - Functions
  - Async
  - Arrays
  - Object destructuring, spread syntax
  - Importing/exporting
- Node, NPM, package.json

Functions

JS function basics

JS functions can look very familiar. In this simple example we see a function which takes a parameter and returns a value.

```
function greet(name) {
  return `Hello, ${name}!`;
}
```

JS functions, alternate syntaxes

JS Functions can be defined using alternative syntaxes.

Function Declaration

```
function greet(name) {
  return `Hello, ${name}!`;
}
```

Function Expression

```
const greet = function (name) {
  return `Hello, ${name}!`;
};
```

Arrow Function

```
const greet = (name) => `Hello, ${name}!`;
```

Anonymous functions

- Unlike programming languages like C#, functions are commonly treated as first class objects in JS.
- Functions are passed to other functions and returned by functions.
- What is the expected output of the code below?

```
const getTransformer = (isUpperCase) => {
  if (isUpperCase) {
    return (inputString) => {
      return inputString.toUpperCase();
    };
  }
  return (inputString) => {
    return inputString.toLowerCase();
  };
};

const transformString = (stringToTransform, transformer) => {
  return transformer(stringToTransform);
};

const message = "Hello Inholland";
console.log(transformString(message, getTransformer(true)));
console.log(transformString(message, getTransformer(false)));
```

`this` keyword in functions

Functions in JS are objects. `this` can refer to the object context.

When the function is declared `this` refers to the object context of that function.

In arrow functions, `this` refers to the parent's object context.

```
function Car(sound) {
  this.sound = sound;
  this.go = () => {
    console.log(this.sound);
  };
}
const car = new Car("vrooooom!");
car.go();
```

Async Code

Async Code in JS

Unlike programming languages like C# and Java which support asynchronous programming but don't always use it, asynchronous programming is *essential* in JS.

Part of the reason Node and JS make such heavy use of asynchronous code is because JS is single threaded. A JS process runs in a single thread and can only perform one operation at a time. If a slow operation (i.e. network or database call) occurs, without async code, the JS thread would be blocked and unable to perform any other operations. This would result in an inefficient and unresponsive application.

Async code in JS allows the single application thread to continue to do other work while the asynchronous operation completes. Once the asynchronous operation completes, the JS thread returns to the location in code and resumes operations there.

Callbacks

There are three main ways to deal with async code: `callbacks`, `promises` and `async`.

Callbacks are the oldest style. They work by passing a function to another function. After the asynchronous action is complete, the callback function is called.

Callback functions are no longer preferred in most cases but are still commonly seen, especially in older libraries.

Callback Example

What is the order of the output? Which console log runs first?

Can you find *both* callback functions in this example?

```
function fetchData(callback) {
  console.log("Fetching data...");
  // Simulate async operation (e.g., API call)
  setTimeout(() => {
    const data = { name: "Alice", age: 25 };
    callback(data); // call the callback when done
  }, 2000);
}

fetchData((result) => {
  console.log("Data received:", result);
});

console.log("At the end of the script.");
```

Promises

The Promise object represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

Promises can be in one of three states:

- `pending` : The promise has not completed.
- `fulfilled` : The promise completed successfully.
- `rejected` : The promise failed.

Promises have three main methods:

- `.then()` : Invoked after successful completion
- `.catch()` : Invoked if an error occurs
- `.finally()` : Invoked on success *or* error

Promise Example 1/2

```
// Simulated API call
function fakeApiCall() {
  return new Promise((resolve, reject) => {
    console.log("📞 Calling the server...");

    setTimeout(() => {
      const success = Math.random() > 0.3; // 70% chance of success

      if (success) {
        resolve("🎉 Data received: { user: 'Alice', age: 25 }");
      } else {
        reject("💥 Server error: something went wrong!");
      }
    }, 2000); // wait 2 seconds to simulate network delay
  });
}
```

Promise Example 2/2

```
// Use the promise
fakeApiCall()
  .then((data) => {
    console.log("✅ Success:", data);
  })
  .catch((error) => {
    console.error("❌ Error:", error);
  })
  .finally(() => {
    console.log("🔚 API call finished (success or fail).");
  });
```

async/await

- is the preferred way to write asynchronous code in JS
- allows us to write asynchronous code that is not deeply nested (i.e. a promise, inside a promise, inside a promise)
- allows us to use `try/catch` blocks to catch errors

To use `async/await` the asynchronous function must be declared `async` and when calling the `async` function, we must `await` it.

`async` functions are really wrappers around `Promise` objects. You can `await` a `Promise`.

async/await example

```
// Async function instead of manual promise
async function getNumber() {
  const num = Math.random();
  if (num > 0.5) {
    return num;
  } else {
    throw "Number too small";
  }
}

async function run() {
  try {
    const result = await getNumber();
    console.log("Success:", result);
  } catch (error) {
    console.error("Error:", error);
  }
}

run();
```

Arrays and Array Methods

From this array, I want a new array with only the names of the sweet fruit. How?

```
const food = [
  {
    name: "banana",
    type: "fruit",
    isSweet: true,
  },
  {
    name: "apple",
    type: "fruit",
    isSweet: true,
  },
  {
    name: "avocado",
    type: "fruit",
    isSweet: false,
  },
  {
    name: "carrot",
    type: "vegetable",
    isSweet: false,
  },
];
```

JS Array Methods

Arrays are a very common data structure in JS.

JS Arrays have several powerful methods that allow for filtering, selection, mapping, etc.

JS Array methods often return new arrays. That makes these methods chainable.

Anonymous functions allow us to easily filter and transform the data in arrays.

It is uncommon to use a `for` loop when dealing with JS arrays.

Array Method Example Chaining

How many items are in `sweetFruitNames` and what are their types?

What is the return type of `.find()`?

What is the meaning of `!!` in the last line?

```
// get the name of all the sweet fruits
const sweetFruitNames = foodItems
  .filter((food) => food.type === "fruit" && food.isSweet === true)
  .map((food) => food.name);

console.log(sweetFruitNames);

// check if the array has a vegetable
const hasVegetable = foodItems.find((food) => food.type === "vegetable");

console.log("has vegetable:", !!hasVegetable);
```

Array Method Reference (see instance methods):

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

Object Destructuring and Spread Syntax

Object Destructuring

Object destructuring allows us to to pluck properties out of objects and turn them into variables.

This is commonly used when methods are returning multiple values or if we want to make our code more readable.

```
const fruit = {
  fruitName: "banana",
  type: "fruit",
  isSweet: true,
};

// destructuring the object
const { fruitName, isSweet } = fruit;

console.log(`The ${fruitName} is sweet: `, isSweet);
// ~> The banana is sweet:  true
```

Array Destructuring

Arrays can also be destructured:

```
const fruits = ["banana", "apple"];
const [banana, apple] = fruits;

console.log(`The first fruit is the ${banana}`);
```

Spread operator

The spread operator (`...`) can be used to flatten objects and arrays and "spread" the properties into new objects or arrays.

```
const baseFruit = { type: "fruit", isHealthy: true };
// spread base fruit properties into apple
const apple = { ...baseFruit, name: "apple" };
console.log(`The type of ${apple.name} is ${apple.type}`);
// ~> The type of apple is fruit
let fruits = [apple];
// spread base fruit elements into fruits array and add a new fruit
fruits = [...fruits, { ...baseFruit, name: "banana" }];
```

Overriding properties

When spreading properties, it is possible to override as long as the spread comes before the new assignment.

```
const baseFruit = { type: "fruit", isHealthy: true };
// spread base fruit and override type when creating a carrot
const carrot = { ...baseFruit, type: "veg", name: "carrot" };
console.log(`The type of ${carrot.name} is ${carrot.type}`);
// ~> The type of carrot is veg
```

Using object destructuring on function arguments and setting defaults

The following pattern of using object destructuring when passing objects as method parameters is common. Default properties can also be set.

```
const printFruit = ({ type = "fruit", isHealthy = true, name } = {}) => {
  console.log(`The ${name} is of type ${type} and is healthy: ${isHealthy}`);
};
printFruit({ name: "mango" });
// ~> The mango is of type fruit and is healthy: true
```

Destructing reference

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring

Importing/exporting

Importing and exporting modules

JS makes makes heavy use of modules. A module is just chunk of code in a separate file that is exported and can be reused in other files. Modules can export data, functions, classes, etc. Any JS object can be exported in a module.

There are two main module systems in JS:

- **Common JS**
- **ECMAScript Modules (ESM)**

Common JS

Common JS was the original module management system for Node. Generally this style is deprecated in favor of ESM (which Node fully supports) but is still commonly seen.

Common JS uses the `module.exports` property to export from a module and `require()` to import that module from another file.

Common JS is typically only used outside the browser context (typically on the server).

Common JS Example

Avoid this style in favor of ESM (which Node supports)!

```
// foo.js
const myFunction = () => {
  console.log("foo!");
};

module.exports = myFunction;
```

```
// index.js
const foo = require("./foo");
foo(); // ~> foo!
```

ECMAScript modules (ESM)

ESM is the official JS module system.

ESM is designed to work both in the browser and on the server.

ESM exports items from a file using the `export` statement and imports them using the `import` statement. Object destructuring is commonly used when importing module items.

ESM export example

Use the `export` keyword to export anything from a file.

Use `export default` to define the default export.

```
// fruit.js
export const fruitType = "fruit";

export const printFruit = ({ type, name }) => {
  console.log(`Fruit type: ${type}, name: ${name}`);
};

export default class Fruit {
  constructor({ type = fruitType, name } = {}) {
    this.type = type;
    this.name = name;
  }
}
```

ESM import example

Use `import` keyword to import items from another module.

The `default export` (the `Fruit` class) does not need to be destructured but the non-default exports do.

```
// index.js
import Fruit, { fruitType, printFruit } from "./fruit.js";
const apple = new Fruit({ type: fruitType, name: "apple" });
printFruit(apple);
// ~> Fruit type: fruit, name: apple
```

JavaScript Essentials