JavaScript Essentials

🎯 Learning Objectives

By the end of this presentation, you will be able to:

- **Write and understand** different JavaScript function syntaxes
- **Handle asynchronous operations** using callbacks, promises, and async/await
- **Manipulate arrays** using modern array methods
- **Use object destructuring** and spread syntax effectively
- **Work with modules** using ES6 import/export
- **Set up and manage** Node.js projects with npm

📋 Prerequisites

Before this presentation, you should be familiar with:

- Basic programming concepts (variables, loops, conditionals)
- HTML and CSS fundamentals
- Basic command line usage
- Text editor usage

🚀 What You'll Learn

- **Functions**
  - Function declarations, expressions, and arrow functions
  - First-class functions and closures
  - The `this` keyword and context
- **Asynchronous Programming**
  - Callbacks, promises, and async/await
  - Event loop and non-blocking operations
  - Error handling in async code
- **Modern JavaScript Features**
  - Array methods (map, filter, reduce, etc.)
  - Object destructuring and spread syntax
  - ES6 modules
- **Node.js Ecosystem**
  - Node.js runtime and npm
  - Package management and project setup

📁 Demo Files

Throughout this presentation, you'll find references to demo files in the `/demos` directory:

- `01_functions.js` - JavaScript Functions
- `02_async.js` - Asynchronous Programming
- `03_arrays.js` - Arrays and Array Methods
- `04_destructuring.js` - Destructuring and Spread Syntax
- `05_modules.js` - ES6 Modules
- `06_node_npm.js` - Node.js and npm

Run demos with: `node demos/[filename].js`

Functions

🧩 Why Functions Matter

Functions are the building blocks of JavaScript applications. They allow you to:

- **Reuse code** and avoid repetition
- **Organize logic** into manageable pieces
- **Pass behavior** as data (first-class functions)
- **Create abstractions** that hide complexity

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}!`;
}

// Usage
console.log(greet("Alice")); // "Hello, Alice!"
```

💡 **Try this in:** `demos/01_functions.js`

JS functions, alternate syntaxes

JS Functions can be defined using alternative syntaxes. Each has its use case:

Function Declaration (Hoisted)

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

Function Expression (Not hoisted)

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

Arrow Function (ES6+)

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

💡 **See examples in:** `demos/01_functions.js`

🤔 When to Use Each Syntax?

Syntax	Use When	Hoisted	Has `this` binding
Function Declaration	General purpose, reusable functions	✅ Yes	✅ Yes
Function Expression	When you need conditional function creation	❌ No	✅ Yes
Arrow Function	Short functions, callbacks, when you want lexical `this`	❌ No	❌ No

Anonymous functions & First-Class Functions

- Unlike programming languages like C#, functions are commonly treated as **first-class objects** in JS
- Functions can be **passed to other functions** and **returned by functions**
- This enables powerful patterns like **higher-order functions**

🧠 Think About This:

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)));
```

💡 Run this example in: `demos/01_functions.js`

🔍 Understanding the Output

```
// Output:
// "HELLO INHOLLAND"
// "hello inholland"
```

What's happening:

1. `getTransformer(true)` returns a function that converts to uppercase
2. `getTransformer(false)` returns a function that converts to lowercase
3. `transformString` calls the returned function with our message
4. This is a **higher-order function** pattern - functions that return functions!

`this` keyword in functions

The `this` keyword refers to the **execution context** of a function.

Key Differences:

- **Regular functions**: `this` refers to the object that calls the function
- **Arrow functions**: `this` refers to the parent's context (lexical scoping)

🎯 Practice: Function Context

Try this yourself:

```
const person = {
  name: "Alice",
  greet: function () {
    console.log(`Hello, I'm ${this.name}`);
  },
  greetArrow: () => {
    console.log(`Hello, I'm ${this.name}`);
  },
};

person.greet(); // What will this output?
person.greetArrow(); // What will this output?
```

Hint: Think about what `this` refers to in each case!

💡 **Practice more in:** `demos/01_functions.js`

Asynchronous Programming

⚡ Why Async Programming?

JavaScript is **single-threaded** - it can only do one thing at a time. Without async programming:

- **Blocking operations** (network calls, file I/O) would freeze the entire application
- **User interfaces** would become unresponsive
- **Performance** would be terrible

Async programming allows JavaScript to:

- **Start operations** and continue with other work
- **Handle multiple tasks** efficiently
- **Keep applications responsive**

🔄 The Event Loop

The event loop allows a single-threaded JS agent to manage asynchronous events.

Parts of the the event loop:

1. **Call Stack** – JavaScript runs functions here in a last-in, first-out (LIFO) order.
2. **Task Queue** – Asynchronous tasks (like setTimeout, I/O, or events) wait here until the stack is empty.
3. **Microtask Queue** – High-priority tasks (like resolved Promises) are processed before the regular task queue.
4. **Event Loop** – Keeps checking if the call stack is empty, then moves tasks from the queues to the stack for execution.

Callbacks

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

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.

⚠️ **Warning:** Callbacks can lead to "callback hell" with nested functions!

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.");
```

🔍 Callback Output Order

```
// Output order:
// 1. "Fetching data..."
// 2. "At the end of the script."
// 3. "Data received: { name: 'Alice', age: 25 }"
```

What's happening:

1. `fetchData` starts and logs "Fetching data..."
2. `setTimeout` schedules the callback for 2 seconds later
3. Script continues and logs "At the end of the script."
4. After 2 seconds, the callback executes

Both callbacks:

- `setTimeout(() => { ... }, 2000)` - arrow function callback
- `(result) => { console.log("Data received:", result); }` - data callback

🚫 Callback Hell Example

```
// This is what we want to avoid!
fetchUser(userId, (user) => {
  fetchUserPosts(user.id, (posts) => {
    fetchPostComments(posts[0].id, (comments) => {
      fetchCommentAuthor(comments[0].authorId, (author) => {
        console.log("Author:", author.name);
      });
    });
  });
});
```

Problems:

- Hard to read and maintain
- Error handling is difficult
- Code becomes deeply nested

💡 **See callback hell and solutions in:** `demos/02_async.js`

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
  });
}
```

💡 Run this example in: `demos/02_async.js`

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).");
  });
```

🔗 Promise Chaining

Promises can be chained to avoid callback hell:

```
fetchUser(userId)
  .then((user) => fetchUserPosts(user.id))
  .then((posts) => fetchPostComments(posts[0].id))
  .then((comments) => fetchCommentAuthor(comments[0].authorId))
  .then((author) => console.log("Author:", author.name))
  .catch((error) => console.error("Error:", error));
```

Benefits:

- Flatter structure
- Better error handling
- More readable code

💡 **See promise chaining in:** `demos/02_async.js`

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();
```

🔄 Converting Callback Hell to async/await

```
// Before (Callback Hell)
fetchUser(userId, (user) => {
  fetchUserPosts(user.id, (posts) => {
    fetchPostComments(posts[0].id, (comments) => {
      fetchCommentAuthor(comments[0].authorId, (author) => {
        console.log("Author:", author.name);
      });
    });
  });
});

// After (async/await)
async function getAuthorInfo(userId) {
  try {
    const user = await fetchUser(userId);
    const posts = await fetchUserPosts(user.id);
    const comments = await fetchPostComments(posts[0].id);
    const author = await fetchCommentAuthor(comments[0].authorId);
    console.log("Author:", author.name);
  } catch (error) {
    console.error("Error:", error);
  }
}
```

💡 See the full conversion in: `demos/02_async.js`

Arrays and Array Methods

🍎 Array Challenge

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,
  },
];
```

💡 **See the solution in:** `demos/03_arrays.js`

💡 Solution: Array Methods

```
const sweetFruitNames = food
  .filter((item) => item.type === "fruit" && item.isSweet)
  .map((item) => item.name);

console.log(sweetFruitNames); // ["banana", "apple"]
```

What we did:

1. **Filtered** for fruits that are sweet
2. **Mapped** to get just the names
3. **Chained** the methods together

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.

🔧 Essential Array Methods

Method	Purpose	Returns	Chainable
`map()`	Transform each element	New array	✅
`filter()`	Select elements that match condition	New array	✅
`reduce()`	Combine elements into single value	Any value	❌
`find()`	Find first matching element	Element or undefined	❌
`some()`	Check if any element matches	Boolean	❌
`every()`	Check if all elements match	Boolean	❌

💡 **See all methods in action in:** `demos/03_arrays.js`

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?

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

// 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);
```

🔍 Understanding the Output

```
// Output:
// ["banana", "apple"]
// has vegetable: true
```

Answers:

- `sweetFruitNames` has **2 items** of type **string**
- `.find()` returns the **first matching element** or **undefined**
- `!!` converts a value to **boolean** (truthy → true, falsy → false)

🎯 Advanced Array Methods

`reduce()` - The Swiss Army Knife

```
const numbers = [1, 2, 3, 4, 5];

// Sum all numbers
const sum = numbers.reduce((acc, num) => acc + num, 0);
console.log(sum); // 15

// Group by type
const grouped = foodItems.reduce((acc, item) => {
  if (!acc[item.type]) acc[item.type] = [];
  acc[item.type].push(item.name);
  return acc;
}, {});

console.log(grouped); // { fruit: ["banana", "apple", "avocado"], vegetable: ["carrot"] }
```

💡 See more reduce examples in: `demos/03_arrays.js`

🎯 Practice: Array Manipulation

Try this yourself:

```
const students = [
  { name: "Alice", grade: 85, subject: "Math" },
  { name: "Bob", grade: 92, subject: "Math" },
  { name: "Charlie", grade: 78, subject: "Science" },
  { name: "Diana", grade: 95, subject: "Math" },
  { name: "Eve", grade: 88, subject: "Science" },
];

// 1. Get all students with grades above 80
// 2. Get the average grade for Math students
// 3. Get a list of unique subjects

// Your code here...
```

💡 See solutions in: `demos/03_arrays.js`

Array Method Reference (see instance methods):

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

Object Destructuring and Spread Syntax

🎯 Why Destructuring?

Object destructuring makes your code:

- **More readable** - extract what you need clearly
- **Less verbose** - no repetitive object.property access
- **More flexible** - easy to rename and set defaults

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
```

💡 **See more examples in:** `demos/04_destructuring.js`

🔧 Destructuring with Renaming

You can rename variables during destructuring:

```
const user = {
  firstName: "Alice",
  lastName: "Johnson",
  age: 25,
};

// Rename firstName to name
const { firstName: name, age } = user;

console.log(`${name} is ${age} years old`);
// ~> Alice is 25 years old
```

Array Destructuring

Arrays can also be destructured:

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

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

🔧 Advanced Array Destructuring

```
const colors = ["red", "green", "blue", "yellow"];

// Skip elements
const [first, , third] = colors;
console.log(first, third); // "red" "blue"

// Rest operator
const [primary, ...others] = colors;
console.log(primary); // "red"
console.log(others); // ["green", "blue", "yellow"]

// Default values
const [a, b, c, d, e = "purple"] = colors;
console.log(e); // "purple"
```

💡 See advanced destructuring in: `demos/04_destructuring.js`

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" }];
```

🔧 Spread with Arrays

```
const fruits = ["apple", "banana"];
const vegetables = ["carrot", "lettuce"];

// Combine arrays
const allFood = [...fruits, ...vegetables];
console.log(allFood); // ["apple", "banana", "carrot", "lettuce"]

// Copy array
const fruitsCopy = [...fruits];
fruitsCopy.push("orange");
console.log(fruits); // ["apple", "banana"]
console.log(fruitsCopy); // ["apple", "banana", "orange"]
```

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 function 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
```

🎯 Practice: Destructuring & Spread

Try this yourself:

```
const user = {
  id: 1,
  name: "Alice",
  email: "alice@example.com",
  preferences: {
    theme: "dark",
    language: "en",
  },
};

// 1. Destructure name and email
// 2. Destructure nested preferences
// 3. Create a new user object with updated preferences
// 4. Use spread to merge two objects

// Your code here...
```

Destructing reference

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

Importing/exporting

📦 Why Modules?

Modules help you:

- **Organize code** into logical units
- **Reuse code** across files
- **Avoid naming conflicts**
- **Control what's public/private**

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
```

🔧 Different Import Styles

```
// Default import
import Fruit from "./fruit.js";

// Named imports
import { fruitType, printFruit } from "./fruit.js";

// Mixed imports
import Fruit, { fruitType, printFruit } from "./fruit.js";

// Namespace import (import everything)
import * as FruitModule from "./fruit.js";
FruitModule.printFruit({ name: "apple", type: "fruit" });

// Rename imports
import { fruitType as type } from "./fruit.js";
```

Node, NPM, package.json

🧐 What is Node.js?

- A **JavaScript runtime** built on **Chrome's V8 engine**
- Allows you to **run JS on the server** 🌐
- Great for **web servers, scripts, and CLI tools**
- Uses **non-blocking, event-driven architecture**

💡 Why Use Node.js?

- Fast execution with **V8 engine**
- **npm ecosystem** with thousands of packages
- Popular in startups & large apps (Netflix, Uber, etc.)

🧰 Installing Node.js

1. Download from nodejs.org
2. Verify installation:

```
node -v
npm -v
```

- `node -v` → check Node version
- `npm -v` → check npm version

📦 What is npm?

- Stands for **Node Package Manager**
- Installs **libraries and tools** for Node.js
- Helps manage **dependencies** in a project
- Comes **bundled with Node.js**

🔧 Installing Packages

- Globally:

```
npm install -g typescript
```

- Locally in project:

```
npm install express
```

- Run scripts defined in **package.json**:

```
npm run start
```

📝 What is `package.json`?

- A **manifest file** for your Node.js project
- Stores:
  - Project info (`name` , `version` , etc.)
  - Dependencies (`dependencies` , `devDependencies`)
  - Scripts (`start` , `test`)
- Created with:

```
npm init
```

📌 Example `package.json`

```
{
  "name": "my-app",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "start": "node index.js",
    "test": "echo \"No tests yet\""
  },
  "dependencies": {
    "express": "^4.18.2"
  }
}
```

- `scripts` → commands you can run with `npm run <name>`
- `dependencies` → libraries needed in production

⚡ Running a Node Project

1. Install dependencies:

```
npm install
```

2. Run your app:

```
npm run start
```

3. Optional: check installed packages:

```
npm list
```

🔍 Dev Dependencies

- Libraries only needed during development:

```
npm install --save-dev nodemon
```

- Example use: auto-reload server during coding

🎯 Practice: Create a Node Project

Try this yourself:

```
# 1. Create a new directory
mkdir my-node-project
cd my-node-project

# 2. Initialize npm project
npm init -y

# 3. Install a dependency
npm install express

# 4. Create index.js
echo "console.log('Hello from Node.js!');" > index.js

# 5. Add a start script to package.json
# 6. Run your project
npm start
```

🎯 Summary

- **Node.js** = JS runtime for server-side
- **npm** = package manager for Node.js
- **package.json** = project manifest, scripts & dependencies
- Install packages & run scripts to manage projects efficiently

📚 Resources

- Node.js Docs
- npm Docs
- npm Package Search

🎉 What You've Learned

- ✅ **Functions**: Different syntaxes, first-class functions, `this` context
- ✅ **Async Programming**: Callbacks, promises, async/await, event loop
- ✅ **Arrays**: Modern methods, chaining, functional programming
- ✅ **Destructuring & Spread**: Object/array destructuring, spread operator
- ✅ **Modules**: ES6 imports/exports, module organization
- ✅ **Node.js**: Runtime, npm, package management

🚀 Next Steps

- **Practice** with the exercises in this presentation
- **Build** start your hackathon project using these concepts
- **Explore** some of the main JavaScript tools we will be using (React, Next.js)
- **Read** the official documentation and MDN Web Docs

Happy coding! 🎯