Detailed Lecture Notes: Writing Your First JavaScript Code

Introduction

In this lecture, we will write our first line of JavaScript code using the browser developer tools. This approach allows us to start quickly without setting up a complete development environment. In the next video, we'll switch to using a code editor.

Opening the Chrome Developer Tools

There are three ways to open the Chrome developer tools:

1. **Keyboard Shortcut**:
    - **Mac**: `Command + Option + J`
    - **Windows**: `Ctrl + Alt + J`
    - This opens the console directly.
2. **Right-Click Method**:
    - Right-click on the webpage and select `Inspect`.
    - This opens the `Elements` tab. Switch to the `Console` tab from here.
3. **Chrome Menu**:
    - Navigate to `Chrome Menu` → `View` → `Developer` → `JavaScript Console`.

Note: Increase the font size using `Command + Plus` (Mac) or `Ctrl + Plus` (Windows) for better visibility.

Using the Console

The JavaScript console allows us to write and test JavaScript code. While it's not used for writing full applications, it is useful for quick experiments and debugging.

Writing Your First JavaScript Code

1. **Alert Function**:
    - Type: `alert("Hello world");` and press `Enter`.
    - This command triggers a popup window displaying "Hello world."
    - **Explanation**:
        - `alert` is a built-in JavaScript function.
        - The text inside the parentheses is called a string, enclosed in double quotes.
2. **Experimenting with JavaScript**:
    - Let's write a variable:

```
let JS = "amazing";
```

- Next, write an if statement:

```
if (JS === "amazing") {
  alert("JavaScript is fun");
}
```

- This will display a popup saying "JavaScript is fun" if `JS` is "amazing."

3. **Changing the Variable**:
   - Change `JS` to "boring":

```
JS = "boring";
```

   - Repeat the if statement:

```
if (JS === "amazing") {
  alert("JavaScript is fun");
}
```

   - This time, no popup will appear since `JS` is not "amazing."
4. **Navigating Previous Commands**:
   - Use the `Up Arrow` key to cycle through previous commands in the console.
5. **Simple Math Operations**:
   - Perform calculations directly:

```
40 + 8 + 23 - 10;
```

   - Press `Enter` to see the result.
   - **Explanation**: This demonstrates how the console can be used as a calculator.

Summary

- The console is a powerful tool for testing and experimenting with JavaScript code.
- Writing `alert("Hello world");` is a simple way to see immediate results of your code.
- Variables can be created using `let`, and conditional logic can be applied with `if` statements.
- The console allows for basic arithmetic operations, showcasing its utility beyond simple alert messages.

Section 2: JavaScript Fundamentals Part 1

Detailed Lecture Notes: Introduction to JavaScript

Introduction

You've written a line or two of JavaScript, but what exactly is JavaScript and what can we do with it? This lecture will set the stage for the rest of the course by answering these questions.

What is JavaScript?

JavaScript is a high-level, object-oriented, multi-paradigm programming language. Let's break down what that means:

1. **Programming Language**:
    - A tool that allows us to write code to instruct a computer to perform tasks.
    - Our main goal is to use JavaScript to write code that executes tasks on a computer.
2. **High-Level Language**:
    - Abstracts away complex details like memory management.
    - Easier to write and learn due to these abstractions.
3. **Object-Oriented**:
    - Based on the concept of objects to store and manage data.
    - We will learn about object-oriented programming (OOP) in detail throughout the course.
4. **Multi-Paradigm**:
    - Supports various programming styles such as imperative (procedural) and declarative (functional).
    - Flexible and versatile, allowing different ways of structuring code.

Role of JavaScript in Web Development

JavaScript is one of the three core technologies of the web, along with HTML and CSS. These technologies work together to create dynamic, interactive websites.

1. **HTML (HyperText Markup Language)**:
    - Responsible for the content of the page (text, images, buttons, etc.).
    - Represents the nouns in our analogy (e.g., a paragraph element `<p>` is a paragraph).
2. **CSS (Cascading Style Sheets)**:
    - Responsible for the presentation of the content (styling and layout).
    - Represents the adjectives in our analogy (e.g., CSS rule `p { color: red; }` describes the paragraph as red).

3. **JavaScript**:
    - The programming language of the web.
    - Adds dynamic and interactive effects to webpages.
    - Represents the verbs in our analogy (e.g., JavaScript code to hide a paragraph).

Capabilities of JavaScript

JavaScript allows developers to:

- Manipulate HTML content and CSS styles.
- Load data from remote servers.
- Build entire web applications (web apps) in the browser.

Real-World Example: Twitter Web Application

Let's look at the Twitter web application to understand JavaScript's capabilities:

1. **Loading Data**:
    - JavaScript loads data in the background, indicated by rotating spinners.
    - Once data is loaded, JavaScript hides the spinners and displays the content.
2. **Dynamic Effects**:
    - Tweet box appears when the tweet button is clicked and hides when clicking outside.
    - User information appears dynamically when hovering over a user.

These examples demonstrate JavaScript's ability to manipulate content, styles, and handle dynamic interactions.

JavaScript's Impact on Modern Web Development

JavaScript has enabled modern web development by allowing developers to create dynamic, interactive web applications that feel like native apps on computers and phones.

1. **JavaScript Libraries and Frameworks**:
    - Tools like React, Angular, and Vue make writing large-scale web applications easier and faster.
    - These tools are based on JavaScript, emphasizing the importance of mastering JavaScript before learning these frameworks.
2. **JavaScript Beyond the Browser**:
    - **Node.js**: Allows JavaScript to run on web servers, enabling backend development.
    - **Native Mobile and Desktop Apps**: Tools like React Native, Ionic, and Electron allow building native applications for phones and desktops using JavaScript.
3. **JavaScript Versions**:
    - Major update in 2015, known as ES2015 or ES6.
    - Yearly updates introduce new features (modern JavaScript).

- Understanding pre-ES2015 JavaScript (ES5) is also important for a comprehensive understanding.

Detailed Lecture Notes: Setting Up and Running JavaScript Code

Introduction

In this lecture, we will learn how to write JavaScript code in a separate file and run it in the browser. We'll start by downloading starter code from GitHub, setting up our project folder, and finally linking JavaScript to an HTML file for execution in the browser.

Getting Started with the Starter Code

1. **Downloading Starter Code**:
   - **Repository**: Visit the provided GitHub repository URL to download the starter code.
   - **Steps**:
     - Click the green "Code" button on GitHub.
     - Select "Download ZIP" to get the starter code.
     - If using a mobile device or smaller screen, scroll to the FAQ section for a download link.
   - **Extracting Files**: Unzip the downloaded file to access the starter code.
2. **Project Structure**:
   - **Folder Organization**:
     - Each section or project has a "final" and "starter" folder.
     - "Final" folder contains the completed code, useful for reference if you encounter issues.
     - "Starter" folder contains the initial code, which is the starting point for the current section.
3. **Setting Up in VS Code**:
   - **Open Folder**:
     - Launch VS Code and select "Open Folder."
     - Choose the "starter" folder from your extracted files.
   - **File Management**:
     - The "index.html" file is the starting point. This file is essential for linking JavaScript code to web pages.

Writing and Running JavaScript

1. **Creating and Linking JavaScript**:
   - **Inline Script**:
     - Insert JavaScript code directly within a `<script>` tag in the HTML file.
     - Example Code:

```
<script>
  let JS = "amazing";
  if (JS === "amazing") {
    alert("JavaScript is fun");
```

```
  }
</script>
```

- **Execution**:
  - Open `index.html` in a web browser to see the JavaScript in action.
  - For example, an alert will display the message "JavaScript is fun."

2. **Displaying Results in the Console**:

- **Using `console.log`**:
  - To display results from JavaScript calculations, use `console.log()`.
  - Example Code:

```
console.log(40 + 8 + 23 - 10);
```

- **Viewing Output**:
  - Open the browser's developer tools (usually F12 or right-click and select "Inspect").
  - Navigate to the "Console" tab to see the output of `console.log()`.

3. **Creating an External JavaScript File**:

- **Steps**:
  - Create a new file named `script.js`.
  - Move JavaScript code from the `<script>` tag in `index.html` to `script.js`.
  - Example Code for `script.js`:

```
let JS = "amazing";
if (JS === "amazing") {
  alert("JavaScript is fun");
}
```

- **Linking External File**:
  - Update the HTML file to link to `script.js`:

```
<script src="script.js"></script>
```

  - Place this `<script>` tag just before the closing `</body>` tag in `index.html`.

4. **Verifying and Troubleshooting**:

- **Check File Location**:
  - Ensure `script.js` is in the same directory as `index.html`.
- **Common Issues**: - Incorrect file paths or spelling errors in the `src` attribute. - JavaScript syntax errors that prevent execution. Certainly! Here are more detailed notes on JavaScript values and

variables:

JavaScript Values and Variables

1. Values

- **Definition**: A value is a fundamental piece of data in programming. It represents a single piece of information.
- **Examples**:
  - **String**: `"Jonas"` (textual data)
  - **Number**: `23` (numerical data)
  - **Boolean**: `true` or `false` (true/false values)
  - **Object**: `{ name: "Jonas", age: 23 }` (complex data structure)
  - **Array**: `[1, 2, 3, 4, 5]` (list of values)
  - **Undefined**: A variable that has been declared but not assigned a value
  - **Null**: A variable that explicitly has no value
- **Displaying Values**: Use `console.log(value)` to output values to the console.

```
console.log("Jonas"); // Outputs: Jonas
console.log(23); // Outputs: 23
```

2. Variables

- **Definition**: Variables are named containers that hold values. They allow you to store and manipulate data in your programs.
- **Declaration and Assignment**:
  - **Syntax**: `let variableName = value;`
  - **Example**:

```
let firstName = "Jonas";
let age = 23;
```

- **Accessing Values**: Refer to the variable name to get the value stored in it.

```
console.log(firstName); // Outputs: Jonas
console.log(age); // Outputs: 23
```

- **Updating Values**: Changing the value of a variable updates all its references.

```
firstName = "Matilda";
console.log(firstName); // Outputs: Matilda
```

3. Variable Naming Conventions

- **camelCase**: Standard naming convention in JavaScript where the first word is lowercase and subsequent words start with an uppercase letter.
  - **Example**: `firstName`, `myFirstJob`
- **snake_case**: An alternative naming convention where words are separated by underscores.
  - **Example**: `first_name`, `my_first_job`
- **UPPER_CASE**: Used for constants that are not intended to change.
  - **Example**: `PI`, `MAX_VALUE`

4. Variable Naming Rules

- **Cannot Start with Numbers**: Variable names cannot begin with a digit.
  - **Invalid**: `3years`
  - **Valid**: `years3`
- **Allowed Characters**: Variables can contain letters, numbers, underscores (_), and dollar signs ($).
  - **Valid**: `first_name`, `age2`, `$value`
  - **Invalid**: `first-name`, `my value` (contains spaces)
- **Reserved Keywords**: Variables cannot use reserved JavaScript keywords.
  - **Examples**: `function`, `new`, `class`
  - **Invalid**: `function = 27;`
  - **Valid**: `_function = 27;` (starting with an underscore or dollar sign)

5. Naming Conventions for Constants

- **Use UPPER_CASE**: Constants that are meant to remain unchanged throughout the program.
  - **Example**: `const PI = 3.14159;`
- **Purpose**: Clearly distinguish constants from variables and improve code readability.

6. Descriptive Naming

- **Importance**: Variable names should be descriptive to indicate their purpose and improve code clarity.
  - **Good Example**: `myFirstJob = "programmer";`
  - **Bad Example**: `job1 = "programmer";`
- **Purpose**: Helps others (and yourself) understand what the variable represents without needing additional context.

7. Error Handling

- **Syntax Errors**: Errors related to incorrect syntax in the code.
  - **Example**: `let 1stName = "Jonas";` (variable name cannot start with a number)
  - **Error Message**: `SyntaxError: Unexpected number`
- **Console Output**: JavaScript errors appear in the console, which helps in debugging.
  - **Example**: `console.log("Hello";` will throw an error `SyntaxError: Unexpected end of input`.

Summary

- **Variables**: Named containers used to store and manipulate data. They simplify code maintenance and updates.
- **Values**: Fundamental units of data that variables store. They can be of various types including strings, numbers, and objects.
- **Naming**: Follow conventions and rules to improve readability and avoid errors.

JavaScript Data Types

In JavaScript, values can be classified into two broad categories: objects and primitive values. For now, we will focus on primitive data types, which are fundamental types that are not objects.

Primitive Data Types

JavaScript has seven primitive data types:

1. **Number**
    - **Description**: Represents both integer and floating-point numbers.
    - **Example**: `23`, `3.14`
    - **Characteristics**: JavaScript does not differentiate between integer and floating-point numbers; all are of the type `number`.
2. **String**
    - **Description**: Represents textual data. Strings are sequences of characters enclosed in quotes.
    - **Example**: `"Hello, World!"`, `'JavaScript'`
    - **Characteristics**: Strings can be defined using single quotes (`'`), double quotes (`"`), or backticks (`` ` ``) for template literals.
3. **Boolean**
    - **Description**: Represents a logical value that can be either `true` or `false`.
    - **Example**: `true`, `false`
    - **Characteristics**: Used for conditional checks and logical operations.
4. **Undefined**
    - **Description**: Represents a variable that has been declared but has not yet been assigned a value.
    - **Example**: `let variable;` (Here, `variable` is `undefined`)
    - **Characteristics**: `undefined` is both the value and the type of an uninitialized variable.
5. **Null**
    - **Description**: Represents an intentional absence of any object value.
    - **Example**: `let value = null;`
    - **Characteristics**: Similar to `undefined` in representing "no value," but used in different contexts.
6. **Symbol** (Introduced in ES2015)
    - **Description**: Represents a unique and immutable value often used as object property keys.
    - **Example**: `Symbol('description')`

- **Characteristics**: Symbols are unique and not meant to be used directly by most developers.

7. **BigInt** (Introduced in ES2020)
   - **Description**: Represents integers that are too large to be represented by the `number` type.
   - **Example**: `9007199254740991n`
   - **Characteristics**: Allows for integer values beyond the safe range of the `number` type.

Dynamic Typing

JavaScript uses dynamic typing, which means:

- **No Manual Type Declaration**: You do not need to explicitly declare the type of a variable. JavaScript determines the type automatically based on the value assigned.
- **Reassigning Values**: You can change the type of a value stored in a variable at any time. For example, a variable initially holding a number can later hold a string.

Example of Dynamic Typing:

```
let variable = 42; // Number
variable = "Hello"; // Now a String
```

Typeof Operator

- **Description**: The `typeof` operator returns a string indicating the type of the unevaluated operand.
- **Syntax**: `typeof value`
- **Example**:

```
console.log(typeof 23); // Outputs: "number"
console.log(typeof "Hello"); // Outputs: "string"
console.log(typeof true); // Outputs: "boolean"
console.log(typeof undefined); // Outputs: "undefined"
console.log(typeof null); // Outputs: "object" (This is a known bug in
JavaScript)
console.log(typeof Symbol("description")); // Outputs: "symbol"
console.log(typeof 9007199254740991n); // Outputs: "bigint"
```

Undefined and Null

- **Undefined**:
  - **Definition**: Value assigned to variables that have been declared but not yet initialized.
  - **Example**: `let x; console.log(x); // Outputs: undefined`
- **Null**:
  - **Definition**: Explicitly assigned value representing the intentional absence of any object value.

- **Example**: `let y = null; console.log(y); // Outputs: null`

Note: The `typeof` operator returns `"object"` for `null`, which is a historical bug and is not corrected for legacy reasons.

Code Commenting

- **Single-Line Comments**: Use `//` for comments that occupy a single line.
  - **Example**:

```
// This is a single-line comment
let x = 10; // Variable x initialized to 10
```

- **Multi-Line Comments**: Use `/*` to start and `*/` to end multi-line comments.
  - **Example**:

```
/* This is a multi-line comment
   It can span multiple lines */
let y = 20;
```

- **Shortcuts in VS Code**:
  - **Single-Line Comment**: `Ctrl + /` (Windows/Linux) or `Cmd + /` (Mac)
  - **Multi-Line Comment**: `Ctrl + Shift + /` (Windows/Linux) or `Cmd + Option + /` (Mac) Here are the detailed notes based on the transcript you provided:

JavaScript Variable Declarations

In JavaScript, you have three primary ways to declare variables: `var`, `let`, and `const`. Each method has its own characteristics and use cases.

1. `let`

- **Purpose**: `let` is used to declare variables that are expected to change their value over time.
- **Example**:

```
let age = 30;
age = 31; // Reassigning the value
```

- **Characteristics**:
  - Allows reassignment of the variable's value.
  - Can be declared without an initial value and assigned later.
  - Suitable for variables whose values are expected to change during the program execution.

2. `const`

- **Purpose**: `const` is used to declare variables that should not change once assigned.
- **Example**:

```
const birthYear = 1991;
// birthYear = 1990;  // This will cause an error
```

- **Characteristics**:
  - Once a value is assigned to a `const` variable, it cannot be changed.
  - Must be initialized at the time of declaration.
  - Ideal for constants or values that are intended to remain unchanged throughout the program.

3. `var`

- **Purpose**: `var` is the traditional way of declaring variables in JavaScript, now mostly replaced by `let` and `const`.
- **Example**:

```
var job = "programmer";
job = "designer"; // Reassigning the value
```

- **Characteristics**:
  - Allows reassignment of the variable's value.
  - Function-scoped, meaning it is accessible within the function it is declared in, rather than block-scoped.
  - Avoid using `var` in modern JavaScript code due to its less predictable scoping behavior.

Best Practices

- **Default to `const`**: Use `const` by default to prevent unintended changes to variables.
- **Use `let`**: Use `let` only when the variable's value needs to be modified.
- **Avoid `var`**: Refrain from using `var` to prevent potential bugs and confusion caused by its function-scoped nature.

Declaring Variables Without `let`, `const`, or `var`

- **Example**:

```
lastName = "Schmedtmann"; // This creates a global variable
console.log(lastName);
```

- **Caution**: Declaring variables without `let`, `const`, or `var` creates properties on the global object (e.g., `window` in browsers). This can lead to unintended side effects and bugs. Always declare variables using

proper keywords.

Summary

- `let`: Use for variables that need to be reassigned.
- `const`: Use for variables that should remain constant.
- `var`: Avoid in favor of `let` and `const`. Here are the detailed notes based on the transcript about basic JavaScript operators:

JavaScript Operators

Operators in JavaScript allow you to transform values, combine multiple values, and perform various operations. They come in several categories, including mathematical, comparison, logical, and assignment operators. This video will cover some fundamental types of operators.

1. Mathematical (Arithmetic) Operators

Mathematical operators are used to perform basic arithmetic operations on values.

- **Addition (`+`)**: Adds two values.

```
let sum = 10 + 5; // sum = 15
```

- **Subtraction (`-`)**: Subtracts one value from another.

```
let difference = 2037 - 1991; // difference = 46
```

- **Multiplication (`*`)**: Multiplies two values.

```
let product = 46 * 2; // product = 92
```

- **Division (`/`)**: Divides one value by another.

```
let quotient = 92 / 6; // quotient = 15.333
```

- **Exponentiation (`**`)**: Raises a number to the power of another number.

```
let power = 2 ** 3; // power = 8
```

- **Example Usage**: Calculating age based on the current year:

```
const now = 2037;
let ageJonas = now - 1991; // ageJonas = 46
let ageSarah = now - 2018; // ageSarah = 19
console.log(ageJonas, ageSarah);
```

2. String Concatenation

The + operator can also be used to concatenate strings (combine them).

- **Example**:

```
let firstName = "Jonas";
let lastName = "Schmedtmann";
let fullName = firstName + " " + lastName; // fullName = 'Jonas Schmedtmann'
console.log(fullName);
```

3. Assignment Operators

Assignment operators are used to assign values to variables and can also combine assignments with operations.

- **Basic Assignment (=)**:

```
let x = 10 + 5; // x = 15
```

- **Add and Assign (+=)**:

```
x += 10; // x = x + 10; x = 25
```

- **Multiply and Assign (*=)**:

```
x *= 4; // x = x * 4; x = 100
```

- **Increment (++)**: Increases the value by one.

```
x++; // x = 101
```

- **Decrement (--)**: Decreases the value by one.

```
```

```
x--; // x = 100
```

4. Comparison Operators

Comparison operators are used to compare values and produce Boolean results (`true` or `false`).

- **Greater Than (`>`)**:

```
let isOlder = ageJonas > ageSarah; // true
```

- **Less Than (`<`)**:

```
let isYounger = ageSarah < 18; // false
```

- **Greater Than or Equal To (`>=`)**:

```
let isFullAge = ageSarah >= 18; // true
```

- **Less Than or Equal To (`<=`)**:

```
let isNotFullAge = ageSarah <= 17; // true if ageSarah is 17 or less
```

5. Operator Precedence

JavaScript follows specific rules for operator precedence (the order in which operators are evaluated).

- **Example**:

```
let result = now - 1991 > now - 2018; // Evaluates to true
```

  Here, the subtraction is performed before the comparison, ensuring correct results.

Summary

- **Mathematical Operators**: Perform arithmetic operations (`+`, `-`, `*`, `/`, `**`).
- **String Concatenation**: Combine strings with `+`.
- **Assignment Operators**: Assign and modify values (`=`, `+=`, `*=`, `++`, `--`).
- **Comparison Operators**: Compare values and return Boolean results (`>`, `<`, `>=`, `<=`).
- **Operator Precedence**: JavaScript determines the order of operations using precedence rules.

JavaScript Operators and Precedence

1. Basic Operators

Operators in JavaScript are used to perform operations on values. There are several categories of operators:

- **Mathematical (Arithmetic) Operators**: Perform arithmetic operations.
- **Comparison Operators**: Compare values and return a Boolean result.
- **Logical Operators**: Combine or invert Boolean values.
- **Assignment Operators**: Assign values to variables.

2. Arithmetic Operators

Arithmetic operators perform basic math operations:

- **Addition (+)**: Adds two values.

```
let sum = 5 + 3; // 8
```

- **Subtraction (-)**: Subtracts one value from another.

```
let difference = 10 - 2; // 8
```

- **Multiplication (*)**: Multiplies two values.

```
let product = 4 * 3; // 12
```

- **Division (/)**: Divides one value by another.

```
let quotient = 12 / 3; // 4
```

- **Exponentiation (**)**: Raises a number to the power of another number.

```
let power = 2 ** 3; // 8
```

3. String Concatenation

The + operator can also be used to concatenate (join) strings:

```
let firstName = "Jonas";
let lastName = "Schmedtmann";
```

```
let fullName = firstName + " " + lastName; // "Jonas Schmedtmann"
```

4. Assignment Operators

Assignment operators assign values to variables and can also modify them:

- **Simple Assignment (`=`)**: Assigns a value to a variable.

```
let x = 10;
```

- **Addition Assignment (`+=`)**: Adds and assigns.

```
x += 5; // Equivalent to x = x + 5
```

- **Subtraction Assignment (`-=`)**: Subtracts and assigns.

```
x -= 3; // Equivalent to x = x - 3
```

- **Multiplication Assignment (`*=`)**: Multiplies and assigns.

```
x *= 2; // Equivalent to x = x * 2
```

- **Division Assignment (`/=`)**: Divides and assigns.

```
x /= 2; // Equivalent to x = x / 2
```

- **Increment (`++`)**: Increases the value by one.

```
x++; // Equivalent to x = x + 1
```

- **Decrement (`--`)**: Decreases the value by one.

```
x--; // Equivalent to x = x - 1
```

5. Comparison Operators

Comparison operators compare values and return a Boolean (`true` or `false`):

- **Equal to (==)**: Checks if two values are equal (type coercion allowed).

```
let isEqual = 5 == "5"; // true
```

- **Strict Equal to (===)**: Checks if two values are equal and of the same type.

```
let isStrictEqual = 5 === "5"; // false
```

- **Not Equal to (!=)**: Checks if two values are not equal.

```
let isNotEqual = 5 != 3; // true
```

- **Strict Not Equal to (!==)**: Checks if two values are not equal or not of the same type.

```
let isStrictNotEqual = 5 !== "5"; // true
```

- **Greater than (>)**: Checks if one value is greater than another.

```
let isGreater = 10 > 5; // true
```

- **Less than (<)**: Checks if one value is less than another.

```
let isLess = 5 < 10; // true
```

- **Greater than or Equal to (>=)**: Checks if one value is greater than or equal to another.

```
let isGreaterOrEqual = 10 >= 10; // true
```

- **Less than or Equal to (<=)**: Checks if one value is less than or equal to another.

```
let isLessOrEqual = 5 <= 10; // true
```

6. Operator Precedence

Operator precedence determines the order in which operators are evaluated. Higher precedence operators are evaluated before lower precedence ones.

Precedence Table Overview:

- **Parentheses (`()`)**: Highest precedence, used to group expressions.
- **Exponentiation (`**`)**: Next highest precedence.
- **Multiplication (`*`), Division (`/`), and Modulus (`%`)**: Next in precedence.
- **Addition (`+`) and Subtraction (`-`)**: Lower precedence than multiplication and division.
- **Comparison Operators (`>`, `<`, `>=`, `<=`)**: Even lower precedence.
- **Assignment Operators (`=`, `+=`, `-=`)**: Lowest precedence among the ones discussed.

Example:

```
let result = 5 + 3 * 2; // 11 because multiplication has higher precedence than
addition
```

Parentheses to Change Order:

```
let result = (5 + 3) * 2; // 16 because parentheses change the order of operations
```

Template Literals in JavaScript

Template literals provide a more powerful and flexible way to work with strings in JavaScript. They simplify string concatenation and allow for easier interpolation and multiline strings.

1. Concatenation Using Template Literals

Instead of using traditional string concatenation with `+`, template literals let you embed expressions directly within the string.

Traditional Concatenation:

```
const firstName = "Jonas";
const birthYear = 1983;
const currentYear = 2023;

const age = currentYear - birthYear;
const job = "teacher";

const introduction =
  "I'm " + firstName + ", a " + age + " years old " + job + ".";
console.log(introduction);
```

Using Template Literals:

```
const firstName = "Jonas";
const birthYear = 1983;
```

```
const currentYear = 2023;

const age = currentYear - birthYear;
const job = "teacher";

const introduction = `I'm ${firstName}, a ${age} years old ${job}.`;
console.log(introduction);
```

- **Backticks** (``) are used to create a template literal.
- **Expressions** inside `${}` are evaluated and inserted into the string.

2. Multiline Strings

Template literals also support multiline strings, making it easier to write and manage complex text.

Before ES6 (using backslashes for new lines):

```
const multilineString =
  "This is a string\n" +
  "that spans multiple lines\n" +
  "using backslashes and n.";
console.log(multilineString);
```

Using Template Literals:

```
const multilineString = `This is a string
that spans multiple lines
using template literals.`;
console.log(multilineString);
```

- Simply press "Enter" to create a new line in the string.

3. Practical Use Cases

1. **Dynamic Content:** Insert variables or expressions directly within the string, making it more readable and maintainable.
2. **Multiline Text:** Write multiline strings without special characters or concatenation.
3. **HTML Templates:** Create and manipulate HTML directly from JavaScript code in a cleaner manner.

Example: HTML Template

```
const title = "Welcome";
const content = "Thank you for visiting our website.";

const htmlTemplate = `
```

```
  <div>
    <h1>${title}</h1>
    <p>${content}</p>
  </div>
`;

document.body.innerHTML = htmlTemplate;
```

- This allows you to create complex HTML structures dynamically with ease.

Summary

Template literals streamline string handling in JavaScript by:

- Allowing inline expressions with `${}`.
- Supporting multiline strings without special characters.
- Enhancing readability and reducing the likelihood of errors.

Using template literals is a modern best practice in JavaScript development and simplifies many common tasks involving strings.

Conditional Statements with `if` and `else` in JavaScript

Conditional statements are fundamental for controlling the flow of execution in a program. They allow you to make decisions and execute different code based on certain conditions.

1. Basic `if` Statement

The `if` statement evaluates a condition and executes a block of code if the condition is true.

Syntax:

```
if (condition) {
  // Code to execute if condition is true
}
```

Example:

```
const age = 19;

if (age >= 18) {
  console.log("Sarah can start her driving license 🚗");
}
```

- Here, `age >= 18` is the condition. If `age` is 18 or more, the message will be logged to the console.

2. Adding `else` Block

The `else` block executes if the `if` condition is false. It provides an alternative action when the condition is not met.

Syntax:

```
if (condition) {
  // Code to execute if condition is true
} else {
  // Code to execute if condition is false
}
```

Example:

```
const age = 15;

if (age >= 18) {
  console.log("Sarah can start her driving license 🚗");
} else {
  const yearsLeft = 18 - age;
  console.log(`Sarah is too young. Wait another ${yearsLeft} years.`);
}
```

- If `age` is less than 18, the `else` block calculates and displays how many years are left until the person can start their driving license.

3. Using `if` Directly with Expressions

You can write the condition directly in the `if` statement without using an intermediate variable.

Example:

```
const age = 15;

if (age >= 18) {
  console.log("Sarah can start her driving license 🚗");
} else {
  const yearsLeft = 18 - age;
  console.log(`Sarah is too young. Wait another ${yearsLeft} years.`);
}
```

4. Conditional Variable Assignment

Variables can be conditionally assigned values based on conditions. This requires defining the variable outside the blocks to ensure it's accessible later.

Example:

```
const birthYear = 1998;
let century;

if (birthYear <= 2000) {
  century = 20;
} else {
  century = 21;
}

console.log(`The person was born in the ${century}th century.`);
```

- `century` is declared outside the `if-else` blocks to ensure it can be used later in the code.

5. Recap

- `if` **Statement**: Executes code if the condition is true.
- `else` **Block**: Executes alternative code if the condition is false (optional).
- **Conditional Variables**: Variables can be assigned values based on conditions and should be declared outside the blocks if they are to be used later.

Coding Challenge: Comparing BMIs with `if`/`else`

In this challenge, you used the `if`/`else` statement to compare the BMIs of two individuals and log a message to the console based on the comparison. Let's break down the solution:

Step-by-Step Solution

1. **Initial Setup**: You have variables for the BMIs of John and Mark:

   ```
   const BMIJohn = 28;
   const BMIMark = 26;
   ```

2. **Comparison Using `if`/`else`**:

   - Use the `if` statement to compare the BMIs:

     ```
     if (BMIMark > BMIJohn) {
       console.log("Mark's BMI is higher than John's!");
     } else {
       console.log("John's BMI is higher than Mark's!");
     }
     ```

     Here, if `BMIMark` is greater than `BMIJohn`, the first message is logged. Otherwise, the `else` block logs the second message.

3. **Testing**:

- By changing the BMI values, you can test different scenarios:

```
// Test case 1
const BMIJohn = 28;
const BMIMark = 26;
// Output: "John's BMI is higher than Mark's!"

// Test case 2
const BMIJohn = 26;
const BMIMark = 28;
// Output: "Mark's BMI is higher than John's!"
```

4. **Using Template Literals**:
   - To include the actual BMI values in the message, use template literals:

```
if (BMIMark > BMIJohn) {
  console.log(`Mark's BMI (${BMIMark}) is higher than John's
(${BMIJohn})!`);
} else {
  console.log(`John's BMI (${BMIJohn}) is higher than Mark's
(${BMIMark})!`);
}
```

   - Template literals (using backticks) allow embedding expressions within a string, making it easy to include dynamic content.

Type Conversion vs. Type Coercion in JavaScript

Type Conversion

Type conversion is the explicit conversion of one data type to another. You perform this manually using built-in functions. Here's how it works:

1. **String to Number Conversion:** To convert a string to a number, use the `Number` function:

```
const inputYear = "1991";
const convertedYear = Number(inputYear); // 1991
```

If you try to add a number to a string without converting the string, it will concatenate the values:

```
console.log(inputYear + 18); // "199118"
```

To properly add 18 to the year, convert the string to a number first:

```
console.log(Number(inputYear) + 18); // 2009
```

If the string cannot be converted to a number, `Number` will return `NaN` (Not-a-Number):

```
console.log(Number("Jonas")); // NaN
console.log(typeof NaN); // "number" (NaN is still of type number)
```

2. **Number to String Conversion:** Convert a number to a string using the `String` function:

```
const num = 23;
const str = String(num); // "23"
```

Type Coercion

Type coercion is the implicit conversion of one data type to another by JavaScript during operations. This happens automatically, and it's important to understand how it works to avoid unexpected behavior:

1. **String Concatenation:** If you use the `+` operator with a string and another type, JavaScript converts the other type to a string and concatenates:

```
console.log("I am " + 23 + " years old"); // "I am 23 years old"
```

2. **Arithmetic Operations:** For operations like subtraction (`-`), multiplication (`*`), and division (`/`), JavaScript converts strings to numbers:

```
console.log("23" - 10 - 3); // 10 (string "23" is converted to number)
console.log("23" * 2); // 46
console.log("23" / 2); // 11.5
```

3. **Examples and Explanation:**
    - **Addition with Strings:**

```
console.log(2 + 3 + 4 + "5"); // "95" (because 2 + 3 + 4 = 9, and "5"
makes it "95")
```

    - **Subtraction and Addition with Strings:**

```
console.log("10" - "4" - "3" - 2 + "5"); // "15"
```

Explanation:

- `"10" - "4"` results in `6`.
- `6 - "3"` results in `3`.
- `3 - 2` results in `1`.
- `1 + "5"` results in `"15"` (string concatenation).

Truthy and Falsy Values in JavaScript

Introduction to Truthy and Falsy Values

Before we can dive into booleans and their behavior, it's essential to understand the concept of truthy and falsy values.

Falsy Values

Falsy values are not exactly false but will become false when converted to a boolean. JavaScript has only five falsy values:

- `0`
- Empty string `""`
- `undefined`
- `null`
- `NaN`

> **Note:** `false` itself is also false, but it is not included in the list of falsy values because it is already a boolean.

Truthy Values

Any value that is not falsy is considered a truthy value. This includes:

- Any number that is not `0`
- Any string that is not an empty string
- Objects and arrays, even if they are empty

Converting Values to Booleans

To see how these values behave in practice, we can use the `Boolean` function to convert different values into booleans.

Example:

```
console.log(Boolean(0)); // false
console.log(Boolean(undefined)); // false
console.log(Boolean("Jonas")); // true
console.log(Boolean({})); // true
console.log(Boolean("")); // false
```

Practical Use of Truthy and Falsy Values

In real-world applications, explicit conversion to boolean using the `Boolean` function is rare. Instead, JavaScript implicitly converts values to booleans in certain contexts.

Implicit Type Coercion to Booleans

JavaScript performs implicit type coercion to booleans in two primary scenarios:

1. Using logical operators
2. In a logical context, such as the condition of an `if-else` statement

Example with `if-else` Statement

Let's consider an example where we want to check if a person has any money:

```
let money = 0;

if (money) {
  console.log("Don't spend it all");
} else {
  console.log("You should get a job");
}
```

Explanation:

- `money` is set to `0`, which is a falsy value.
- In the `if-else` condition, `money` is implicitly converted to a boolean.
- Since `0` is falsy, the `else` block is executed, logging "You should get a job".

Changing the value of `money` to a truthy value, such as `100`:

```
let money = 100;

if (money) {
  console.log("Don't spend it all");
} else {
  console.log("You should get a job");
}
```

- Now, `100` is a truthy value.
- The `if` block is executed, logging "Don't spend it all".

Checking for Defined Variables

Another common use case for truthy and falsy values is to check if a variable is defined:

```
let height;

if (height) {
  console.log("YAY! Height is defined");
} else {
  console.log("Height is undefined");
}
```

Explanation:

- Initially, `height` is `undefined`, a falsy value.
- The `else` block is executed, logging "Height is undefined".

Assigning a value to `height`:

```
let height = 100;

if (height) {
  console.log("YAY! Height is defined");
} else {
  console.log("Height is undefined");
}
```

- `height` is now `100`, a truthy value.
- The `if` block is executed, logging "YAY! Height is defined".

Potential Issue with Falsy Values

A potential issue arises when a falsy value like `0` is a valid value:

```
let height = 0;

if (height) {
  console.log("YAY! Height is defined");
} else {
  console.log("Height is undefined");
}
```

- `height` is `0`, which is falsy.
- The `else` block is executed, incorrectly logging "Height is undefined".

Equality Operators in JavaScript

When we want to check if two values are equal, rather than comparing them to see if one is greater or less than the other, we use equality operators. Here's a detailed look at how they work:

Strict Equality Operator (`===`)

The strict equality operator checks if two values are exactly the same, without performing type coercion. If the values and their types match, the result is `true`; otherwise, it's `false`.

Example:

```
let age = 18;
if (age === 18) {
  console.log("You just became an adult!");
}
```

In this example, the condition `age === 18` is true, so the message "You just became an adult!" is logged to the console.

- **Usage without Curly Braces:** If the `if` statement has only one line of code, you can omit the curly braces `{}`.

Testing in the Console:

```
console.log(18 === 18); // true
console.log(18 === 19); // false
```

Loose Equality Operator (`==`)

The loose equality operator checks if two values are equal, performing type coercion if necessary. This means it converts the values to a common type before making the comparison.

Example:

```
console.log(18 == "18"); // true
console.log(18 === "18"); // false
```

In this example, `18 == '18'` returns true because the loose equality operator converts the string '18' to the number 18 before comparing. However, `18 === '18'` returns false because strict equality checks both value and type, and they are different types.

General Rule: Avoid using the loose equality operator to prevent unexpected bugs. Always use the strict equality operator (`===`).

Converting User Input

When working with user input, it’s often a string. To ensure accurate comparisons, convert the string to the desired type before using strict equality.

Example:

```
const favorite = Number(prompt("What's your favorite number?"));
if (favorite === 23) {
  console.log("Cool! 23 is an amazing number!");
}
```

Here, the `Number` function converts the user input from a string to a number, ensuring the strict equality check works as expected.

Adding More Conditions with `else if`

You can add multiple conditions in an `if else` statement using `else if`.

Example:

```
const favorite = Number(prompt("What's your favorite number?"));
if (favorite === 23) {
  console.log("Cool! 23 is an amazing number!");
} else if (favorite === 7) {
  console.log("7 is also a cool number!");
} else if (favorite === 9) {
  console.log("9 is also a cool number!");
} else {
  console.log("Number is not 23, 7, or 9");
}
```

This example checks multiple conditions one after the other. If none of the conditions are met, the final `else` block is executed.

The Different Operator (`!==`)

Just as we have operators for equality, we also have operators to check if values are different.

- **Strict Inequality Operator (`!==`):** Checks if two values are not exactly the same, without type coercion.
- **Loose Inequality Operator (`!=`):** Checks if two values are not equal, performing type coercion if necessary.

Example:

```
const favorite = Number(prompt("What's your favorite number?"));
if (favorite !== 23) {
  console.log("Why not 23?");
}
```

Here, `favorite !== 23` checks if the user's favorite number is not 23, logging a message if it's not.

Summary

- Use `===` for strict equality checks to avoid type coercion.
- Use `!==` for strict inequality checks.
- Avoid using `==` and `!=` to prevent unexpected behavior due to type coercion.
- Convert values to the desired type manually before making comparisons for accurate results.

Boolean Logic in JavaScript

Boolean logic is a crucial aspect of computer science that utilizes true and false values to solve complex logical problems. This branch of logic employs logical operators to combine true and false values, similar to how arithmetic operators combine numeric values.

Key Logical Operators

1. **AND Operator (&&)**: Returns true if both operands are true.
2. **OR Operator (||)**: Returns true if at least one of the operands is true.
3. **NOT Operator (!)**: Inverts the Boolean value of the operand.

Understanding Logical Operators with Examples

Example Scenario

Let's consider a scenario where we have two Boolean variables:

- `A`: Sarah has a driver's license.
- `B`: Sarah has good vision.

These variables can either be true or false.

Truth Tables

1. **AND Operator (&&)**
    - Returns true only if both operands are true.
    - Truth Table:

A	B	A && B
true	true	true
true	false	false
false	true	false
false	false	false

2. **OR Operator (||)**
    - Returns true if at least one operand is true.
    - Truth Table:

A	B	A	B
true	true	true	

A	B	A	B
true	false	true	
false	true	true	
false	false	false	

3. **NOT Operator (!)**
   - Inverts the Boolean value.
   - Truth Table:

A	!A
true	false
false	true

Practical Example with Age

Let's use a practical example to better understand these concepts.

1. **Initial Setup**

```
let age = 16;
let A = age >= 20; // false
let B = age < 30; // true
```

2. **Using Logical Operators**
   - **NOT Operator**

```
let notA = !A; // true
```

   - **AND Operator**

```
let andOperation = A && B; // false
```

   - **OR Operator**

```
let orOperation = A || B; // true
```

   - **Combining Operators**

```
let notAAndB = !A && B; // true
let aOrNotB = A || !B; // false
```

Example Code Implementation

1. **Setting up the Age Variable**

```
let age = 16;
```

2. **Defining Boolean Variables**

```
let A = age >= 20; // false
let B = age < 30; // true
```

3. **Performing Logical Operations**

```
let notA = !A; // true
let andOperation = A && B; // false
let orOperation = A || B; // true
let notAAndB = !A && B; // true
let aOrNotB = A || !B; // false
```

- The **AND operator** returns true only if both operands are true.
- The **OR operator** returns true if at least one operand is true.
- The **NOT operator** inverts the Boolean value.

Understanding Logical Operators in JavaScript

Let's dive into using logical operators in JavaScript with practical examples. We'll use Boolean variables to illustrate how `AND`, `OR`, and `NOT` operators work.

Example Scenario

We have two Boolean variables:

- `hasDriversLicense`: Represents if Sarah has a driver's license.
- `hasGoodVision`: Represents if Sarah has good vision.

We'll start by setting both variables to `true` and then explore different logical operations.

Creating Boolean Variables

```
let hasDriversLicense = true; // Variable A
let hasGoodVision = true; // Variable B
```

Using Logical Operators

1. **AND Operator (`&&`)**

   The `AND` operator returns `true` only if both operands are `true`.

   ```
   console.log(hasDriversLicense && hasGoodVision); // true && true => true
   ```

   Change `hasGoodVision` to `false`:

   ```
   hasGoodVision = false;
   console.log(hasDriversLicense && hasGoodVision); // true && false => false
   ```

2. **OR Operator (`||`)**

   The `OR` operator returns `true` if at least one operand is `true`.

   ```
   console.log(hasDriversLicense || hasGoodVision); // true || false => true
   ```

3. **NOT Operator (`!`)**

   The `NOT` operator inverts the Boolean value of the operand.

   ```
   console.log(!hasDriversLicense); // !true => false
   ```

Decision Making Example

Let's determine if Sarah should drive. We'll create a new Boolean variable `shouldDrive` based on the conditions.

```
let shouldDrive = hasDriversLicense && hasGoodVision;

if (shouldDrive) {
  console.log("Sarah is able to drive");
} else {
  console.log("Someone else should drive");
}
```

With `hasGoodVision` set to `false`, the output will be:

```
Someone else should drive
```

Change both variables to `true`:

```
hasDriversLicense = true;
hasGoodVision = true;
```

Now the output will be:

```
Sarah is able to drive
```

Adding Another Variable

Let's add another Boolean variable `isTired` and include it in our decision-making process.

```
let isTired = true; // Variable C
```

Now, Sarah should drive only if she has a driver's license, good vision, and is not tired.

```
shouldDrive = hasDriversLicense && hasGoodVision && !isTired;

if (shouldDrive) {
  console.log("Sarah is able to drive");
} else {
  console.log("Someone else should drive");
}
```

With `isTired` set to `true`, the output will be:

```
Someone else should drive
```

Change `isTired` to `false`:

```
isTired = false;
```

Now the output will be:

```
Sarah is able to drive
```

Summary

- The `AND` operator (`&&`) returns `true` only if all operands are `true`.
- The `OR` operator (`||`) returns `true` if at least one operand is `true`.
- The `NOT` operator (`!`) inverts the Boolean value.

Learning About the Switch Statement in JavaScript

The switch statement in JavaScript provides an alternative way to write a complicated if/else statement when you need to compare a single value to multiple options. This can make your code more readable and organized.

Example Scenario

We have a variable `day` that represents a day of the week, and we want to map each day to a specific activity.

Initial Setup

```
let day = "Monday"; // Hardcoded for learning purposes
```

Switch Statement Syntax

The switch statement compares the value of `day` to multiple cases and executes the corresponding code block.

```
switch (day) {
  case "Monday":
    console.log("Plan course structure");
    console.log("Go to coding meetup");
    break;
  case "Tuesday":
    console.log("Prepare theory videos");
    break;
  case "Wednesday":
  case "Thursday":
    console.log("Write code examples");
    break;
  case "Friday":
    console.log("Record videos");
    break;
  case "Saturday":
  case "Sunday":
    console.log("Enjoy the weekend");
    break;
```

```
    default:
      console.log("Not a valid day");
}
```

Explanation

- **case 'Monday':** If `day` is 'Monday', it logs the activities for Monday and then breaks out of the switch statement.
- **case 'Tuesday':** Logs the activity for Tuesday.
- **case 'Wednesday':** and **case 'Thursday':** Both log the same activity for Wednesday and Thursday.
- **case 'Friday':** Logs the activity for Friday.
- **case 'Saturday':** and **case 'Sunday':** Both log the same activity for the weekend.
- **default:** If `day` doesn't match any case, it logs 'Not a valid day'.

The `break` statement ensures that the switch statement exits after executing the matched case. Without `break`, the code will continue to execute the following cases, which is usually not desired.

Comparing to If/Else Statement

The same logic can be implemented using an if/else statement.

```
if (day === "Monday") {
  console.log("Plan course structure");
  console.log("Go to coding meetup");
} else if (day === "Tuesday") {
  console.log("Prepare theory videos");
} else if (day === "Wednesday" || day === "Thursday") {
  console.log("Write code examples");
} else if (day === "Friday") {
  console.log("Record videos");
} else if (day === "Saturday" || day === "Sunday") {
  console.log("Enjoy the weekend");
} else {
  console.log("Not a valid day");
}
```

Key Points

- **Switch Statement:** Provides a cleaner and more readable syntax for comparing a single value to multiple options.
- **If/Else Statement:** Can handle more complex logical conditions but may become less readable with many options.
- **Strict Comparison:** The switch statement uses strict comparison (`===`) to match the cases.

Statements vs. Expressions in JavaScript

Understanding the difference between statements and expressions is crucial in JavaScript as it helps in writing and comprehending code more effectively.

Expressions

An expression is any valid unit of code that resolves to a value. Here's a breakdown of some examples:

- **Arithmetic Expression:** `3 + 4` (resolves to 7)
- **Literal Expression:** `42` (resolves to the number 42)
- **Boolean Expression:** `true && false` (resolves to false)

Expressions can be more complex, involving multiple operations and functions, but they always produce a single value.

```
// Examples of expressions
3 + 4; // 7
42; // 42
true && false; // false
```

Statements

A statement is a piece of code that performs an action. It does not necessarily produce a value and often forms the building blocks of a program, like sentences in a language.

- **Variable Declaration:** `let x = 5;`
- **If Statement:** `if (x > 0) { console.log('x is positive'); }`
- **Loop Statement:** `for (let i = 0; i < 10; i++) { console.log(i); }`

Statements execute code but do not resolve to a value directly.

```
// Examples of statements
let x = 5;
if (x > 0) {
  console.log("x is positive");
}
for (let i = 0; i < 10; i++) {
  console.log(i);
}
```

Combining Expressions and Statements

Statements can contain expressions, but expressions themselves cannot contain statements. For instance, an if statement can include expressions as conditions or inside its block:

```
let y = 10;
if (y > 5) {
```

```
  y = y * 2; // 'y * 2' is an expression
}
```

Important Differences

1. **End with Semicolon:** Statements often end with a semicolon (`;`), signifying the end of an action.
2. **Placement in Code:** Expressions can be used where values are expected (like in variable assignments), while statements form the structure of the code.

Expressions in Template Literals

Template literals in JavaScript can only include expressions within the `${}` syntax, not statements.

```
let age = 30;
console.log(`I am ${2037 - 1991} years old.`); // Valid
```

Attempting to include a statement in a template literal results in an error:

```
// This will throw an error
console.log(`I am ${if (age > 18) { 'adult'; } else { 'child'; }}`);
```

Practical Implications

Knowing where expressions and statements can be used helps in avoiding syntax errors and writing code that behaves as expected. For example, in a template literal, you should use expressions to embed dynamic values.

Summary

- **Expressions:** Produce values, can be used within other expressions, suitable for places expecting a value.
- **Statements:** Perform actions, often end with a semicolon, used to control the flow and structure of the program.

Conditional (Ternary) Operator in JavaScript

The conditional (ternary) operator is a concise way to perform conditionals in JavaScript, enabling you to write an if/else statement all on one line. This operator is especially useful for simple conditions and expressions.

Syntax and Basic Usage

The ternary operator has the following syntax:

```
condition ? expressionIfTrue : expressionIfFalse;
```

This can be broken down into three parts:

1. **Condition:** The test to evaluate (e.g., `age >= 18`).
2. **Expression if True:** The code to execute if the condition is true.
3. **Expression if False:** The code to execute if the condition is false.

Here's a practical example:

```
let age = 23;
age >= 18
  ? console.log("I like to drink wine 🍷")
  : console.log("I like to drink water 💧");
```

If the condition `age >= 18` is true, it logs "I like to drink wine 🍷". If false, it logs "I like to drink water 💧".

Storing the Result in a Variable

Since the ternary operator produces a value, you can store this value in a variable:

```
let age = 23;
let drink = age >= 18 ? "wine" : "water";
console.log(drink); // Outputs: wine
```

Without the ternary operator, you would need a more verbose if/else statement:

```
let drink;
if (age >= 18) {
  drink = "wine";
} else {
  drink = "water";
}
console.log(drink); // Outputs: wine
```

Using the Ternary Operator in Template Literals

The ternary operator can be used within template literals, which expect expressions:

```
let age = 23;
console.log(`I like to drink ${age >= 18 ? "wine" : "water"}.`); // Outputs: I
like to drink wine.
```

This allows for concise conditional logic within strings.

Key Takeaways

1. **Conciseness:** The ternary operator allows for more concise and readable code compared to if/else statements, especially for simple conditions.
2. **Expressions:** Since the ternary operator is an expression, it produces a value that can be used directly or stored in variables.
3. **Template Literals:** It can be effectively used within template literals for conditional string construction.
4. **Not a Replacement:** The ternary operator is not a replacement for if/else statements when handling complex conditions or multiple lines of code. It is best used for simple, quick decisions.

Example Comparison

Ternary Operator:

```
let age = 23;
let drink = age >= 18 ? "wine" : "water";
console.log(drink); // Outputs: wine
```

If/Else Statement:

```
let age = 23;
let drink;
if (age >= 18) {
  drink = "wine";
} else {
  drink = "water";
}
console.log(drink); // Outputs: wine
```

The ternary operator makes the code more concise and easier to read for simple conditions, whereas if/else statements are better suited for more complex logic.

Understanding JavaScript Releases and Versions

Importance of JavaScript Releases

Knowing about JavaScript versions and releases is crucial for web developers. It helps in staying updated with the latest features and understanding discussions within the developer community.

History of JavaScript

1. **Early Days**:
    - **1995**: Netscape Navigator, the dominant browser, hired Brendan Eich to create the first version of JavaScript, called "Mocha," in just 10 days.
    - **1996**: Mocha was renamed "Livescript" and then "JavaScript" to attract Java developers, even though JavaScript and Java are unrelated.
2. **Browser Wars**:

- Microsoft launched Internet Explorer in 1996, which copied JavaScript but called it "JScript" for legal reasons.

3. **Standardization**:
    - **1997**: JavaScript was standardized by ECMA, resulting in ECMAScript 1 (ES1).
    - ECMAScript refers to the standard, while JavaScript is the language implemented in browsers.

Key Releases and Updates

1. **ES5 (2009)**:
    - Introduced many new features and is well-supported across browsers today.
2. **ES6 (2015)**:
    - The biggest update, introducing numerous features that modernized the language. Also called ES2015.
    - Changed to an annual release cycle, leading to ES2016 (ES7), ES2017, and so on.

Backward and Forward Compatibility

- **Backward Compatibility**:
    - JavaScript engines can run code written in older versions without issues, adhering to the principle of "not breaking the web."
- **Forward Compatibility**:
    - Modern browsers cannot run code from future versions, which is why transpiling and polyfilling are necessary.

Using Modern JavaScript Today

1. **Development Phase**:
    - Use the latest version of Google Chrome to ensure compatibility with the latest JavaScript features.
2. **Production Phase**:
    - Convert modern JavaScript code to ES5 using tools like Babel to ensure it runs in older browsers.

Browser Support

- ES5 is supported in all modern browsers, including Internet Explorer 9.
- ES6 and later versions (ES6+) are well-supported in most modern browsers.
- Future features are often implemented by browsers at stage 3 of the proposal process, even before becoming official.

Key Takeaways

- **JavaScript Evolution**: From Mocha in 1995 to ES6 in 2015, JavaScript has evolved significantly.
- **Backward Compatibility**: Ensures old code runs in modern browsers.
- **Forward Compatibility**: Addressed using transpiling and polyfilling.
- **Modern JavaScript**: Use the latest browsers for development and transpile code for production.

Section 3: JavaScript fundamentals - Part 2

Activating Strict Mode in JavaScript

Introduction to Strict Mode

Before continuing with JavaScript, we should activate a special mode called "Strict Mode." This mode helps in writing more secure and error-free JavaScript code.

Setting Up the Environment

1. **Folder Structure**
    - Have a folder ready for this section, which contains the starter files from the GitHub repository provided at the beginning of the course.
    - Copy these files to your computer and open them in VS Code.
2. **HTML and Script Files**
    - Ensure the HTML file links to the JavaScript file, as this is already set up in the starter files.
    - Open the working folder in VS Code.

Activating Strict Mode

- **How to Activate**
    - At the beginning of your script, add `"use strict";`. This line must be the first statement (excluding comments) in your script or function.

```
"use strict";
// Your code here
```

- **Scope of Strict Mode**
    - Strict mode can be applied to an entire script or specific functions/blocks.
    - It is recommended to apply it to the entire script for consistency and security.

Benefits of Strict Mode

1. **Error Prevention**
    - Helps developers avoid accidental errors by forbidding certain actions.
    - Makes bugs easier to identify by creating visible errors.
2. **Example of Strict Mode**

```
"use strict";

let hasDriversLicense = false;
const passTest = true;

if (passTest) hasDriversLicense = true;

if (hasDriversLicense) console.log("I can drive");
```

- **Without Strict Mode**
  - Typos in variable names might go unnoticed, causing bugs.
- **With Strict Mode**
  - Throws errors for undefined variables, helping catch mistakes early.

Demonstrating Strict Mode with an Example

1. **Initial Setup**
   - Create variables `hasDriversLicense` and `passTest`.

```
"use strict";

let hasDriversLicense = false;
const passTest = true;
```

2. **Intentional Bug**
   - Introduce a typo in the variable name.

```
if (passTest) hasDriverLicense = true; // Missing 's'
```

3. **Expected Output**
   - Log a message if the driver has a license.

```
if (hasDriversLicense) console.log("I can drive");
```

4. **Observing the Error**
   - Without strict mode, the typo might not be caught, causing the program to fail silently.
   - With strict mode, an error is thrown, indicating the undefined variable.

Reserved Keywords in Strict Mode

- **Future-Proofing JavaScript**
  - Strict mode reserves certain keywords for future language features.
  - Attempting to use these reserved keywords will throw errors.

```
"use strict";

const interface = "Audio Interface"; // Throws error
const private = 123; // Throws error
```

Conclusion

- **Importance of Strict Mode**
  - Activating strict mode should be a standard practice for writing secure and bug-free JavaScript code.
  - It helps catch common mistakes and makes debugging easier.
- **Moving Forward**
  - All scripts in this course will assume strict mode is enabled.
  - Always start your scripts with `"use strict";` to benefit from its features.

With strict mode out of the way, we can now proceed to learning about functions in JavaScript.

Functions in JavaScript

Introduction

Functions are a fundamental building block in JavaScript applications. They allow us to write reusable code chunks, similar to how variables store values. Functions can hold multiple lines of code, enabling us to execute the same code block multiple times.

Declaring a Function

To declare a function:

1. Use the `function` keyword.
2. Define a function name.
3. Use parentheses `()` and curly braces `{}` to create the function body.

Example:

```
function logger() {
  console.log("My name is Jonas");
}
```

Invoking a Function

To invoke a function, write the function name followed by parentheses. This process is called calling, running, or invoking the function.

Example:

```
logger(); // My name is Jonas
logger();
logger();
```

Functions with Parameters

Functions can accept parameters (inputs) and return values (outputs). Parameters are placeholders for values that will be provided when the function is called.

Example:

```
function fruitProcessor(apples, oranges) {
  const juice = `Juice with ${apples} apples and ${oranges} oranges.`;
  return juice;
}

const appleJuice = fruitProcessor(5, 0);
console.log(appleJuice); // Juice with 5 apples and 0 oranges

const appleOrangeJuice = fruitProcessor(2, 4);
console.log(appleOrangeJuice); // Juice with 2 apples and 4 oranges
```

Understanding Parameters and Arguments

- **Parameters**: Variables defined in the function declaration (`apples`, `oranges`).
- **Arguments**: Actual values passed to the function when called (`5`, `0`).

Example:

```
fruitProcessor(5, 0); // 5 and 0 are arguments
```

Return Values

Functions can return values using the `return` keyword. This value can be captured in a variable for further use.

Example:

```
function add(a, b) {
  return a + b;
```

```
}

const sum = add(2, 3);
console.log(sum); // 5
```

Reusability and DRY Principle

Functions promote code reusability and help follow the DRY (Don't Repeat Yourself) principle. Instead of writing the same code multiple times, encapsulate it in a function and call it when needed.

Summary

- Functions are reusable code blocks.
- Declare functions with `function` keyword, a name, parameters (optional), and a body.
- Call functions with their name followed by parentheses.
- Functions can accept parameters and return values.
- Use functions to keep code DRY and maintainable.

Practical Example

```
function logger() {
  console.log("My name is Jonas");
}

function fruitProcessor(apples, oranges) {
  const juice = `Juice with ${apples} apples and ${oranges} oranges.`;
  return juice;
}

// Calling functions
logger();
const juice1 = fruitProcessor(5, 0);
console.log(juice1);

const juice2 = fruitProcessor(2, 4);
console.log(juice2);
```

By understanding and using functions effectively, you can write cleaner, more maintainable JavaScript code. Functions are a core concept in JavaScript, essential for building robust applications.

Lecture: Function Declarations and Expressions in JavaScript

In this lecture, we'll explore the different ways of writing functions in JavaScript. Each type of function works slightly differently, but they share many similarities. Let's dive into the details.

Function Declarations

Function declarations are one way to define a function in JavaScript. You use the `function` keyword followed by a function name. Here's an example of a function declaration to calculate age based on a given birth year:

```
function calcAge1(birthYear) {
  return 2037 - birthYear;
}
```

Key points about function declarations:

- **Keyword**: Use the `function` keyword.
- **Naming**: You must name the function.
- **Parameters**: Parameters are placeholders for inputs, treated as local variables within the function.
- **Return**: Use the `return` keyword to output the result.

Calling the function and storing the result:

```
const age1 = calcAge1(1991);
console.log(age1); // Outputs: 46
```

Function Expressions

Function expressions involve creating an anonymous function (a function without a name) and assigning it to a variable. Here's the same age calculation function written as a function expression:

```
const calcAge2 = function (birthYear) {
  return 2037 - birthYear;
};
```

Key points about function expressions:

- **Anonymous Function**: No name is given to the function.
- **Variable Assignment**: The function is assigned to a variable, which then holds the function.
- **Usage**: Call the function using the variable name.

Calling the function expression and storing the result:

```
const age2 = calcAge2(1991);
console.log(age2); // Outputs: 46
```

Differences Between Function Declarations and Expressions

1. **Hoisting**:

- Function declarations are hoisted, meaning they can be called before they are defined in the code.
- Function expressions are not hoisted. Attempting to call them before their definition results in an error.

Example of hoisting with function declaration:

```
console.log(calcAge1(1991)); // Outputs: 46

function calcAge1(birthYear) {
  return 2037 - birthYear;
}
```

Example of no hoisting with function expression:

```
console.log(calcAge2(1991)); // Error: Cannot access 'calcAge2' before
initialization

const calcAge2 = function (birthYear) {
  return 2037 - birthYear;
};
```

2. **Personal Preference**:
   - Some developers prefer function declarations for their straightforward structure.
   - Others prefer function expressions for a more structured code where functions are defined before being called.

Regardless of preference, it's important to know both types as they have their place in JavaScript development.

Key Takeaways

- **Function Declarations**: Defined with the `function` keyword and a name, can be called before they are defined.
- **Function Expressions**: Anonymous functions assigned to variables, cannot be called before their definition.
- **Usage**: Both types are used in different contexts, and knowing both is essential for effective JavaScript programming.

Lecture: Arrow Functions in JavaScript

In this lecture, we'll explore the third type of function in JavaScript, introduced in ES6: the arrow function. Arrow functions are a special form of function expression that are shorter and faster to write.

Arrow Functions

Arrow functions provide a concise syntax to define functions. Here's an example of converting a function expression to an arrow function for calculating age:

```
const calcAge3 = (birthYear) => 2037 - birthYear;
```

Key points about arrow functions:

- **Concise Syntax**: Arrow functions use `=>` to separate parameters and function body.
- **Implicit Return**: For single-line functions, the return is implicit, meaning there's no need to use the `return` keyword.
- **No Curly Braces**: For single-line functions, curly braces `{}` are not needed.

Using the arrow function:

```
const age3 = calcAge3(1991);
console.log(age3); // Outputs: 46
```

Handling Multiple Lines and Parameters

Arrow functions can handle multiple lines of code and multiple parameters, but the syntax changes slightly.

Multiple Lines

For functions with more than one line of code, use curly braces `{}` and the `return` keyword:

```
const yearsUntilRetirement = (birthYear) => {
  const age = 2037 - birthYear;
  const retirement = 65 - age;
  return retirement;
};

console.log(yearsUntilRetirement(1991)); // Outputs: 19
```

Multiple Parameters

When an arrow function has multiple parameters, enclose them in parentheses `()`:

```
const yearsUntilRetirement = (birthYear, firstName) => {
  const age = 2037 - birthYear;
  const retirement = 65 - age;
  return `${firstName} retires in ${retirement} years.`;
};

console.log(yearsUntilRetirement(1991, "Jonas")); // Outputs: Jonas retires in 19
```

```
years.
console.log(yearsUntilRetirement(1980, "Bob")); // Outputs: Bob retires in 8
years.
```

Comparison with Function Declarations and Expressions

1. **Conciseness**:
    - Arrow functions are more concise and easier to write, especially for simple one-liners.
2. **Implicit Return**:
    - Single-line arrow functions implicitly return the value.
3. **No `this` Binding**:
    - Arrow functions do not have their own `this` keyword. This difference can be crucial and will be covered in more depth later.
4. **Personal Preference**:
    - Arrow functions are excellent for simple tasks, while traditional functions might be better for more complex logic.

When to Use Arrow Functions

While arrow functions are convenient, they are not always the best choice. Use them primarily for:

- Simple, one-liner functions.
- Callbacks, such as in array methods (`map`, `filter`, `reduce`).

For more complex functions or when `this` binding is required, stick to function declarations or expressions.

Key Takeaways

- **Arrow Functions**: Concise syntax, implicit return for single-line functions, no `this` keyword.
- **Usage**: Ideal for simple tasks and callbacks, but not a one-size-fits-all solution.
- **Syntax**: `const funcName = (param1, param2) => { /* function body */ }`.

Arrow functions are a powerful addition to JavaScript, streamlining the syntax for simple functions while retaining the flexibility of function expressions. Use them wisely to write clean and efficient code.

Lecture: Functions Calling Other Functions

In this lecture, we'll delve deeper into the concept of calling one function from within another. This is a common practice in JavaScript, though it can sometimes be challenging for beginners to grasp.

Example: Fruit Processor

We'll use the fruit processor example to illustrate this concept. Previously, we had a function that received a number of apples and oranges and produced juice. Now, we'll add a step where the fruits are cut into smaller

pieces before making the juice.

Step-by-Step Implementation

1. **Cutting Fruit Function**:
    - Create a function that cuts a fruit into multiple pieces.

```
function cutFruitPieces(fruit) {
  return fruit * 4;
}
```

2. **Modified Fruit Processor**:
    - Use the `cutFruitPieces` function within the fruit processor to cut the received apples and oranges into smaller pieces.

```
function fruitProcessor(apples, oranges) {
  const applePieces = cutFruitPieces(apples);
  const orangePieces = cutFruitPieces(oranges);

  const juice = `Juice with ${applePieces} pieces of apple and
${orangePieces} pieces of orange.`;
  return juice;
}
```

3. **Calling the Functions**:
    - Call the `fruitProcessor` function with example values and log the result to the console.

```
console.log(fruitProcessor(2, 3)); // Outputs: Juice with 8 pieces of apple
and 12 pieces of orange.
```

Detailed Analysis of Data Flow

- **Calling `fruitProcessor`**:
    - When `fruitProcessor(2, 3)` is called, the arguments `2` and `3` replace the parameters `apples` and `oranges`.
- **Inside `fruitProcessor`**:
    - The `cutFruitPieces` function is called with `apples` (2) and `oranges` (3) as arguments.
    - `cutFruitPieces(2)` returns `8` (since 2 * 4 = 8).
    - `cutFruitPieces(3)` returns `12` (since 3 * 4 = 12).

- **Building the Juice String**:
    - The resulting pieces are used to construct the juice string: `"Juice with 8 pieces of apple and 12 pieces of orange."`

Advantages of This Approach

1. **Reusability**:
    - The `cutFruitPieces` function can be reused for cutting any fruit, making the code more modular and easier to maintain.
2. **DRY Principle (Don't Repeat Yourself)**:
    - If the cutting logic changes (e.g., cut into 3 pieces instead of 4), we only need to update the `cutFruitPieces` function. This reduces redundancy and potential for errors.
3. **Clarity and Maintenance**:
    - Separating logic into smaller functions makes the code more readable and easier to debug.

Practice and Understanding

- **Exercise**:
    - Create another example where one function calls another.
    - Analyze the data flow and understand how parameters and arguments are passed between functions.

Conclusion

Calling functions within other functions is a powerful technique in JavaScript. It promotes code reusability and adherence to the DRY principle. As you practice, you'll become more comfortable with when and how to create and call nested functions.

Key Takeaways

- **Nested Functions**: One function calling another is a common and useful pattern.
- **Modularity**: Break down complex logic into smaller, reusable functions.
- **Maintainability**: Changes in one part of the code (e.g., cutting logic) require updates in a single place, making the code easier to maintain.

Understanding and using nested functions effectively is a crucial skill in JavaScript development. With practice, this concept will become second nature.

Functions Review

Overview

In this section, we reviewed important concepts about functions in JavaScript to ensure a solid understanding before moving on to other topics.

Years Until Retirement Function

Initial Arrow Function

Here's the initial function written as an arrow function:

```
const yearsUntilRetirement = (birthYear, firstName) => {
  const age = 2037 - birthYear;
  const retirement = 65 - age;
  return `${firstName} retires in ${retirement} years.`;
};
```

Converting to Regular Function Expression

We converted it into a regular function expression:

```
const yearsUntilRetirement = function (birthYear, firstName) {
  const age = 2037 - birthYear;
  const retirement = 65 - age;
  return `${firstName} retires in ${retirement} years.`;
};
```

Calling Functions Within Functions

Extracting Age Calculation

We extracted the age calculation into a separate function:

```
const calcAge = function (birthYear) {
  return 2037 - birthYear;
};
```

Using `calcAge` in `yearsUntilRetirement`

We then updated `yearsUntilRetirement` to use `calcAge`:

```
const yearsUntilRetirement = function (birthYear, firstName) {
  const age = calcAge(birthYear);
  const retirement = 65 - age;
  return `${firstName} retires in ${retirement} years.`;
};
```

Handling Already Retired Cases

To handle cases where the person is already retired:

```
const yearsUntilRetirement = function (birthYear, firstName) {
  const age = calcAge(birthYear);
  const retirement = 65 - age;

  if (retirement > 0) {
    return `${firstName} retires in ${retirement} years.`;
  } else {
    return `${firstName} has already retired.`;
  }
};
```

Testing the Function

We tested the function with different inputs to ensure correctness:

```
console.log(yearsUntilRetirement(1991, "Jonas")); // Jonas retires in 19 years.
console.log(yearsUntilRetirement(1950, "Mike")); // Mike has already retired.
```

Key Concepts and Best Practices

Function Parameters and Arguments

- Parameters are placeholders used within the function.
- Arguments are actual values passed to the function when it is called.
- Parameters with the same name in different functions are independent and act as local variables.

Return Statement

- The `return` statement exits the function immediately.
- Any code after the `return` statement will not be executed.

DRY Principle

- The **Don't Repeat Yourself** principle emphasizes reusing code to avoid duplication.
- Extracting common functionality into separate functions helps maintain and update the code efficiently.

Types of Functions

1. **Function Declaration**: Can be used before they are declared in the code.

```
function calcAge(birthYear) {
  return 2037 - birthYear;
}
```

2. **Function Expression**: Stored in a variable.

```
const calcAge = function (birthYear) {
  return 2037 - birthYear;
};
```

3. **Arrow Function**: A compact form for writing functions.

```
const calcAge = (birthYear) => 2037 - birthYear;
```

Structure of a Function

- **Name**: Identifies the function.
- **Parameters**: Input placeholders.
- **Function Body**: Contains the code to be executed.
- **Return Statement**: Outputs a value and exits the function.

Calling a Function

- Use parentheses to invoke the function.
- Pass arguments to provide input data.
- The function returns a value, which can be stored in a variable.

Console.log vs. Return

- `console.log`: Prints a message to the console for debugging.
- `return`: Outputs a value from the function and terminates its execution.

Summary

Functions are fundamental building blocks in JavaScript. Understanding how to declare, invoke, and utilize functions efficiently is crucial for writing clea n and maintainable code. Practice writing your own functions and using them within other functions to reinforce these concepts.

Arrays in JavaScript

Introduction

Arrays are one of the fundamental data structures in JavaScript. They allow us to store multiple values in a single variable, making it easy to manage and manipulate lists of items. In this section, we'll explore how to create, access, and manipulate arrays.

Storing Multiple Values

Without arrays, storing multiple values would require creating multiple variables, which is not efficient:

```
const friend1 = "Michael";
const friend2 = "Steven";
const friend3 = "Peter";
```

Imagine having to do this for 10 or more friends! Instead, we can use an array to bundle all these values together.

Creating Arrays

To create an array, we use square brackets `[]`:

```
const friends = ["Michael", "Steven", "Peter"];
console.log(friends); // Output: ['Michael', 'Steven', 'Peter']
```

Another way to create an array is by using the `Array` constructor:

```
const years = new Array(1991, 1984, 2008, 2020);
console.log(years); // Output: [1991, 1984, 2008, 2020]
```

Accessing Array Elements

Arrays in JavaScript are zero-based, meaning the first element is at index 0:

```
console.log(friends[0]); // Output: 'Michael'
console.log(friends[2]); // Output: 'Peter'
```

You can also get the length of an array:

```
console.log(friends.length); // Output: 3
```

To get the last element of an array:

```
console.log(friends[friends.length - 1]); // Output: 'Peter'
```

Mutating Arrays

Even though arrays declared with `const` cannot be reassigned, their elements can still be changed:

```
friends[2] = "Jay";
console.log(friends); // Output: ['Michael', 'Steven', 'Jay']
```

However, you cannot reassign the entire array:

```
// This will throw an error
friends = ["Bob", "Alice"];
```

Arrays with Mixed Data Types

Arrays can hold values of different types, including other arrays:

```
const jonas = ["Jonas", "Schmedtmann", 2037 - 1991, "teacher", friends];
console.log(jonas); // Output: ['Jonas', 'Schmedtmann', 46, 'teacher', ['Michael',
'Steven', 'Jay']]
```

Array Exercise

Let's use our `calcAge` function to calculate ages for a list of birth years:

```
const calcAge = function (birthYear) {
  return 2037 - birthYear;
};

const years = [1990, 1967, 2002, 2010, 2018];
const age1 = calcAge(years[0]);
const age2 = calcAge(years[1]);
const age3 = calcAge(years[years.length - 1]);

console.log(age1, age2, age3); // Output: 47, 70, 19
```

We can also store these ages in a new array:

```
const ages = [
  calcAge(years[0]),
  calcAge(years[1]),
  calcAge(years[years.length - 1]),
];
console.log(ages); // Output: [47, 70, 19]
```

Summary

Arrays are a powerful and flexible data structure in JavaScript. They allow you to store and manipulate multiple values efficiently. Understanding arrays and their operations is crucial for effective JavaScript programming. In the next section, we'll delve into more advanced operations you can perform on arrays to make them even more useful.

JavaScript Array Methods

JavaScript provides built-in functions that can be directly applied to arrays, called methods. These methods perform various operations on arrays, and understanding them is essential for effective JavaScript programming.

Common Array Methods

1. `push()`
   - **Purpose**: Adds elements to the end of an array.
   - **Usage**:

```
let friends = ["Michael", "Peter", "Steven"];
friends.push("Jay"); // ['Michael', 'Peter', 'Steven', 'Jay']
```

   - **Returns**: The new length of the array.

```
let newLength = friends.push("Jay"); // 4
```

2. `unshift()`
   - **Purpose**: Adds elements to the beginning of an array.
   - **Usage**:

```
friends.unshift("John"); // ['John', 'Michael', 'Peter', 'Steven']
```

   - **Returns**: The new length of the array.

3. `pop()`
   - **Purpose**: Removes the last element of the array.
   - **Usage**:

```
friends.pop(); // ['Michael', 'Peter']
```

   - **Returns**: The removed element.

```
let popped = friends.pop(); // 'Steven'
```

4. `shift()`
   - **Purpose**: Removes the first element of the array.
   - **Usage**:

```
friends.shift(); // ['Peter', 'Steven']
```

   - **Returns**: The removed element.

5. `indexOf()`
   - **Purpose**: Finds the index of a specific element in the array.
   - **Usage**:

```
let index = friends.indexOf("Steven"); // 1
```

   - **Returns**: The index of the element or -1 if not found.

6. `includes()`
   - **Purpose**: Checks if an array contains a specific element.
   - **Usage**:

```
let hasSteven = friends.includes("Steven"); // true
let hasBob = friends.includes("Bob"); // false
```

   - **Returns**: `true` if the element is found, otherwise `false`.
   - **Example with strict equality check**:

```
friends.push(23);
friends.includes("23"); // false
friends.includes(23); // true
```

   - **Using `includes()` in a conditional**:

```
if (friends.includes("Steven")) {
  console.log("You have a friend called Steven");
}
```

These methods provide powerful ways to manipulate and interact with arrays in JavaScript. As you progress, you'll encounter many more array methods that offer additional functionality and flexibility.

JavaScript Objects

Objects in JavaScript provide a way to store and manage data as key-value pairs, where each key (or property) is associated with a value. This contrasts with arrays, which use ordered indices to access their elements. Objects are particularly useful for grouping related data and for scenarios where you want to reference values by names rather than positions.

Creating an Object

You can create an object using curly braces `{}` and define key-value pairs within them.

Syntax:

```
let objectName = {
  key1: value1,
  key2: value2,
  // More key-value pairs
};
```

Example:

```
let jonas = {
  firstName: "Jonas",
  lastName: "Schmedtmann",
  age: 30,
  job: "Teacher",
  friends: ["Michael", "Peter", "Steven"],
};
```

In this example, `jonas` is an object with five properties:

- `firstName` with the value `'Jonas'`
- `lastName` with the value `'Schmedtmann'`
- `age` with the value `30`
- `job` with the value `'Teacher'`
- `friends` with an array of strings `['Michael', 'Peter', 'Steven']`

Accessing Object Properties

You can access the properties of an object using either dot notation or bracket notation.

1. **Dot Notation**:

```
console.log(jonas.firstName); // 'Jonas'
console.log(jonas.age); // 30
```

2. **Bracket Notation**:

```
console.log(jonas["firstName"]); // 'Jonas'
console.log(jonas["age"]); // 30
```

Bracket notation is useful when the property names are dynamic or not valid identifiers (e.g., contain spaces or special characters).

Modifying Object Properties

You can also modify the properties of an object using either dot notation or bracket notation.

Example:

```
jonas.age = 31; // Using dot notation
jonas["job"] = "Senior Teacher"; // Using bracket notation
```

Adding New Properties

To add new properties to an object, use the same notation.

Example:

```
jonas.hobby = "Cooking"; // Using dot notation
jonas["married"] = true; // Using bracket notation
```

Deleting Properties

You can delete properties from an object using the `delete` operator.

Example:

```
delete jonas.age; // Removes the age property
```

Key Points

- **Order Doesn't Matter**: Unlike arrays, the order of properties in an object does not affect how you access them.

- **Use Cases**: Use objects when you need to group related data together and access it by named keys rather than by position.

Summary

Objects are a fundamental part of JavaScript, allowing you to group and manage data effectively. They are ideal for scenarios where you need to label and retrieve values by name rather than position. Understanding objects and their properties is crucial for effective JavaScript programming and data management.

Retrieving and Modifying Data in JavaScript Objects

Accessing Object Properties

To retrieve data from an object, you can use two notations:

1. **Dot Notation**:

   ```
   console.log(jonas.lastName); // 'Schmedtmann'
   ```

   Dot notation is straightforward and clean but only works when the property name is a valid identifier (no spaces or special characters) and is known beforehand.

2. **Bracket Notation**:

   ```
   console.log(jonas["lastName"]); // 'Schmedtmann'
   ```

   Bracket notation allows for more flexibility:

   - You can use any valid expression inside the brackets, such as variables or concatenated strings.
   - It is useful when the property name is dynamic or not a valid identifier.

   Example:

   ```
   let nameKey = "lastName";
   console.log(jonas[nameKey]); // 'Schmedtmann'
   ```

Practical Use Cases for Bracket Notation

- **Dynamic Property Names**: When you need to compute property names or retrieve them based on user input or other variables.

  Example:

  ```
  let property = prompt("What do you want to know about Jonas?");
  console.log(jonas[property]); // Outputs the value of the chosen property
  ```

If the property does not exist, it returns `undefined`.

- **Handling Undefined Properties**:

```
if (jonas[property]) {
  console.log(jonas[property]);
} else {
  console.log("Wrong request");
}
```

Modifying Object Properties

You can add or update properties using either notation:

1. **Dot Notation**:

```
jonas.location = "Portugal";
jonas.twitterHandle = "@Jonasschmedtman";
```

2. **Bracket Notation**:

```
jonas["location"] = "Portugal";
jonas["twitterHandle"] = "@Jonasschmedtman";
```

Both methods work similarly and can handle any property name.

Challenge: Dynamic Sentence Creation

Create a sentence dynamically using properties from the `jonas` object:

Objective: Write a sentence like "Jonas has three friends and his best friend is called Michael."

1. **Retrieve Values**:
   - Name: `jonas.firstName`
   - Number of Friends: `jonas.friends.length`
   - Best Friend: `jonas.friends[0]`

2. **Code**:

```
let sentence = `${jonas.firstName} has ${jonas.friends.length} friends and
his best friend is called ${jonas.friends[0]}.`;
console.log(sentence);
```

Operator Precedence

- **Dot Notation** and **Bracket Notation** both have high precedence and are evaluated left to right.
  - `jonas.friends.length` is evaluated as: first `jonas.friends`, then `.length`.
  - `jonas.friends[0]` is evaluated as: first `jonas.friends`, then `[0]`.

Summary

- **Dot Notation**: Simple and clean for static property names.
- **Bracket Notation**: Flexible for dynamic or non-standard property names, and expressions.
- Both notations can be used for accessing and modifying object properties, but bracket notation is essential when dealing with dynamic property names. Great lecture on object methods! Let's break down the main points and concepts:

Object Methods

1. **Functions as Object Properties**:
   - Just like other data types (strings, numbers, arrays), functions can be values in key-value pairs in objects.
   - When a function is used as a property in an object, it is called a **method**.
2. **Creating Methods**:
   - Methods are defined similarly to function expressions but are assigned as properties of an object.
   - Example:

```
const jonas = {
  firstName: "Jonas",
  birthYear: 1991,
  hasDriversLicense: true,
  calcAge: function () {
    return 2037 - this.birthYear;
  },
};
```

3. **Accessing Methods**:
   - Methods can be accessed using dot notation or bracket notation.
   - Example:

```
jonas.calcAge(); // Using dot notation
jonas["calcAge"](); // Using bracket notation
```

4. **Using `this` Keyword**:
   - The `this` keyword inside a method refers to the object that is calling the method.

- Example:

```
const jonas = {
  firstName: "Jonas",
  birthYear: 1991,
  calcAge: function () {
    return 2037 - this.birthYear;
  },
};
console.log(jonas.calcAge()); // Uses `this.birthYear` inside the
method
```

5. **Avoiding Repetition**:

- Using `this` helps avoid repeating values and keeps code DRY (Don't Repeat Yourself).
- If you hard-code values instead of using `this`, you might need to update multiple places if the value changes.

6. **Efficiency**:

- Instead of recalculating values multiple times, compute once and store it in the object.
- Example:

```
const jonas = {
  firstName: "Jonas",
  birthYear: 1991,
  calcAge: function () {
    this.age = 2037 - this.birthYear;
    return this.age;
  },
};
```

7. **Challenge: `getSummary` Method**:

- Create a method that summarizes an object's data in a string format.
- Example:

```
const jonas = {
  firstName: "Jonas",
  birthYear: 1991,
  job: "teacher",
  hasDriversLicense: true,
  calcAge: function () {
    this.age = 2037 - this.birthYear;
    return this.age;
  },
  getSummary: function () {
    return `${this.firstName} is a ${this.calcAge()} year old ${
```

```
        this.job
      } and he ${
        this.hasDriversLicense ? "has" : "has no"
      } a driver's license.`;
    },
  };
  console.log(jonas.getSummary());
```

8. **Arrays as Objects**:
    - Arrays in JavaScript are a special type of object with built-in methods (e.g., `push`, `pop`).
    - This concept of methods applies to arrays just like it does to other objects.

Summary

- Methods are functions assigned as properties in objects.
- The `this` keyword allows methods to access and modify the object they belong to.
- Use `this` to keep code DRY and efficient.
- Arrays are also objects and can have methods.

Feel free to ask if you have any questions or need further clarification! Great explanation on the `for` loop! Here's a summary of the key points from your explanation:

For Loop in JavaScript

Purpose: Loops automate repetitive tasks, allowing you to run the same code multiple times without repeating it manually.

Structure

1. **Initialization**: Set up a counter variable.
    - Example: `let rep = 1;`
2. **Condition**: Specify the condition that must be true for the loop to continue.
    - Example: `rep <= 10;`
3. **Update**: Modify the counter variable after each iteration.
    - Example: `rep++` (or `rep = rep + 1`)

Example

```
for (let rep = 1; rep <= 10; rep++) {
  console.log(`Lifting weights repetition ${rep}`);
}
```

- **Initialization**: `let rep = 1;`

- **Condition**: `rep <= 10`
- **Update**: `rep++`

Explanation

- **Initialization**: Sets the starting point of the loop counter.
- **Condition**: Checked before each iteration. If `true`, the loop continues; if `false`, the loop stops.
- **Update**: Increment or modify the counter to eventually meet the loop's exit condition.

Notes:

- **Do Not Repeat Yourself Principle**: Loops help avoid repetitive code.
- **Counter Variable**: The variable `rep` in the example tracks the number of iterations and is used to customize the output dynamically.

Advanced Tip: You can start the counter at any value and set different conditions to suit your needs, such as starting from `5` or going up to `30`.

For Loop in JavaScript

The `for` loop is a fundamental construct in programming used to repeat a block of code a specific number of times. Here's a breakdown of its syntax and use cases based on the provided content:

Basic Structure of a For Loop

```
for (initialization; condition; increment) {
  // Code to be executed
}
```

- **Initialization**: Sets up the counter variable.
- **Condition**: Specifies when to stop the loop.
- **Increment**: Updates the counter variable after each iteration.

Example: Looping Through an Array

Consider an array named `Jonas`:

```
const Jonas = ["Jonas", "Skaag", 46, ["ice cream", "chocolate"], true];
```

To log each element of this array, use the following loop:

```
for (let i = 0; i < Jonas.length; i++) {
  console.log(Jonas[i]);
}
```

- **Initialization**: `let i = 0` starts the counter at 0.
- **Condition**: `i < Jonas.length` ensures the loop runs until `i` is less than the length of the array.
- **Increment**: `i++` increases the counter by 1 each time the loop runs.

Output: Logs each element of the `Jonas` array.

Handling Array Length Dynamically

Instead of hard-coding the array length, use `Jonas.length` to handle dynamic arrays:

```
for (let i = 0; i < Jonas.length; i++) {
  console.log(Jonas[i]);
}
```

This approach ensures that the loop adapts to changes in the array length.

Example: Creating a New Array of Types

To create an array of the types of elements from the `Jonas` array:

```
const types = [];

for (let i = 0; i < Jonas.length; i++) {
  types[i] = typeof Jonas[i];
}

console.log(types); // Output: ["string", "string", "number", "object", "boolean"]
```

Alternatively, using `push`:

```
const types = [];

for (let i = 0; i < Jonas.length; i++) {
  types.push(typeof Jonas[i]);
}

console.log(types); // Output: ["string", "string", "number", "object", "boolean"]
```

Practical Example: Calculating Ages

Given an array of birth years, calculate the ages and store them in a new array:

```
const years = [1991, 2007, 1969, 2020];
const ages = [];
const currentYear = 2037;
```

```
for (let i = 0; i < years.length; i++) {
  ages.push(currentYear - years[i]);
}

console.log(ages); // Output: [46, 30, 68, 17]
```

Using `continue` and `break` Statements

- `continue`: Skips the current iteration and continues with the next one.

  ```
  for (let i = 0; i < Jonas.length; i++) {
    if (typeof Jonas[i] !== "string") continue;
    console.log(Jonas[i]); // Logs only strings
  }
  ```

- `break`: Exits the loop entirely.

  ```
  for (let i = 0; i < Jonas.length; i++) {
    if (typeof Jonas[i] === "number") break;
    console.log(Jonas[i]); // Logs elements until a number is found
  }
  ```

Summary

- **Initialization**: Start counter variable.
- **Condition**: Define loop continuation.
- **Increment**: Update counter each iteration.
- **Dynamic Length**: Use `.length` property.
- **`continue` and `break`**: Control loop flow for specific conditions.

This overview should help you understand how to effectively use `for` loops to iterate over arrays and handle different scenarios in JavaScript. In this lecture, the focus is on two key programming concepts related to loops:

1. Looping Over an Array Backwards

To loop through an array in reverse order, follow these steps:

1. **Initialize the Counter**: Start the counter at the last index of the array. For an array named `Jonas`, the last index can be accessed using `Jonas.length - 1`.
2. **Loop Condition**: Continue the loop as long as the counter is greater than or equal to zero.
3. **Decrement the Counter**: Decrease the counter by one after each iteration using `i--`.

Here's an example of the loop in action:

```
const Jonas = ["Jonas", "Michael", "Sarah", "Lisa", "Tom"];

for (let i = Jonas.length - 1; i >= 0; i--) {
  console.log(Jonas[i]);
}
```

This will print:

```
Tom
Lisa
Sarah
Michael
Jonas
```

2. Nested Loops

Nested loops involve placing one loop inside another. This is useful for scenarios where you need to perform multiple iterations within each iteration of the outer loop.

Example Scenario

Suppose you have three different exercises, and you want to perform five repetitions for each exercise.

1. **Outer Loop**: Handles the exercises.
2. **Inner Loop**: Handles the repetitions for each exercise.

Here's how you can set up the nested loops:

```
for (let exercise = 1; exercise <= 3; exercise++) {
  console.log(`Exercise ${exercise}`);

  for (let rep = 1; rep <= 5; rep++) {
    console.log(`Lifting weights repetition ${rep}`);
  }
}
```

This will print:

```
Exercise 1
Lifting weights repetition 1
Lifting weights repetition 2
Lifting weights repetition 3
Lifting weights repetition 4
Lifting weights repetition 5
Exercise 2
```

```
Lifting weights repetition 1
Lifting weights repetition 2
Lifting weights repetition 3
Lifting weights repetition 4
Lifting weights repetition 5
Exercise 3
Lifting weights repetition 1
Lifting weights repetition 2
Lifting weights repetition 3
Lifting weights repetition 4
Lifting weights repetition 5
```

Key Takeaways

- **Backward Looping**: Start from the end of the array and move to the beginning. Initialize the counter with the last index and decrement it.
- **Nested Loops**: Useful for iterating through complex data structures or scenarios that require multiple levels of iteration.

This practice helps in understanding how loops can be controlled and utilized for more complex operations.

Introduction to the `while` Loop

The `while` loop in JavaScript is used for scenarios where you want to repeat a block of code as long as a specified condition is true. Unlike the `for` loop, which requires initialization, condition, and increment/decrement in one line, the `while` loop focuses solely on the condition.

Comparison: `for` Loop vs. `while` Loop

- **`for` Loop**: Suitable when you know beforehand how many times you want to repeat a block of code. It's great for iterating over arrays or ranges with a known count.
- **`while` Loop**: More flexible, used when the number of iterations is not known ahead of time. It continues as long as the condition remains true.

Example: Weightlifting Exercise with `while` Loop

You can replicate the behavior of the `for` loop using a `while` loop. Here's how you would translate the `for` loop for weightlifting repetitions into a `while` loop:

`for` Loop Example

```
for (let rep = 1; rep <= 10; rep++) {
  console.log(`Lifting weights repetition ${rep}`);
}
```

`while` Loop Example

```
let rep = 1;
while (rep <= 10) {
  console.log(`Lifting weights repetition ${rep}`);
  rep++;
}
```

Implementing a `while` Loop for a Dice Roll

Problem Statement

Roll a dice until you get a 6. This problem does not have a predefined number of iterations, making the `while` loop a suitable choice.

Solution

1. **Initialize the Dice Roll**: Generate a random number between 1 and 6.
2. **Condition**: Continue rolling the dice until the result is 6.
3. **Update**: Re-generate the dice roll value in each iteration.

Here's the implementation:

```
let dice;
do {
  dice = Math.trunc(Math.random() * 6) + 1;
  console.log(`You rolled a ${dice}`);
} while (dice !== 6);

console.log("Loop is about to end");
```

Explanation

- `Math.random()` generates a number between 0 and 1.
- `Math.random() * 6` scales it to a range between 0 and 6.
- `Math.trunc()` removes the decimal part.
- `+ 1` adjusts the range to 1 to 6.

The `do...while` loop ensures that the dice is rolled at least once, and continues rolling until a 6 is rolled.

Summary

- `for` **Loop**: Best when the number of iterations is known.
- `while` **Loop**: Best for cases where the number of iterations is unknown and depends on a condition being met.

Section 7: JS in the browser : DOM and Event fundamentals

Introduction to the "Guess My Number!" Project

Welcome to the first project in this section, where we'll build a fun game called "Guess My Number!" This project is inspired by retro 80s games and has a simple goal: guess a secret number between 1 and 20. Here's a quick overview of the game features and functionalities:

1. **Guessing the Number**: You type a number into an input field and click "Check!" to see if your guess is correct.
2. **Feedback Messages**: You receive feedback on whether your guess is too high, too low, or correct.
3. **Score Tracking**: The game starts with a score of 20, which decreases by 1 with each incorrect guess.
4. **Highscore**: The game keeps track of the highest score achieved.
5. **Play Again**: You can reset the game while keeping the highscore by clicking the "Again!" button.

Project Setup

1. **Download Starter Files**: Begin by downloading the starter files from the GitHub repository.
2. **Open Project Folder**: Open the folder containing the starter files.
3. **File Overview**:
    - **Prettier Configuration**: Ensures consistent code formatting.
    - **Script.js**: Initially empty, where we'll write our JavaScript code.
    - **Style.css**: Contains the styling for the project.
    - **index.html**: The main HTML file containing the structure of the game.

Exploring the HTML Structure

The HTML file contains various elements such as the input field, buttons, and messages. Each element has specific class names that we'll use to select and manipulate them using JavaScript.

Selecting Elements in JavaScript

To interact with the HTML elements, we need to select them using JavaScript. We use the `document.querySelector` method to do this. Let's see an example:

Example: Selecting and Logging an Element

HTML:

```
<p class="message">Start guessing...</p>
```

JavaScript:

```
document.querySelector(".message");
console.log(document.querySelector(".message").textContent); // Logs "Start
guessing..."
```

- **Explanation**:
  - `document.querySelector('.message')` selects the element with the class `message`.
  - `.textContent` retrieves the text content of the selected element.

Setting Up Live Server

To see our changes in real-time, we'll use a live server:

1. **Open Terminal**: Open the terminal in your code editor.
2. **Navigate to Project Folder**: Ensure you're in the project directory.
3. **Start Live Server**: Type `live-server` and hit Enter. This opens the project in a new browser tab and automatically refreshes it whenever you save changes.

First DOM Manipulation

Let's start by logging the text content of the message element to the console.

1. **Write the Code**:

```
console.log(document.querySelector(".message").textContent);
```

2. **Save and Check**:
   - Save your changes.
   - Check the browser console to see the logged text content.

Introduction to DOM Manipulation

In this section, we're going to make JavaScript interact with a webpage for the first time through DOM Manipulation.

What is the DOM?

DOM stands for Document Object Model. It is a structured representation of HTML documents, allowing JavaScript to access and manipulate HTML elements and styles. This enables us to change text, HTML attributes, and CSS styles dynamically.

Key Concepts of the DOM

1. **Document Object**: The DOM is automatically created by the browser as soon as the HTML page loads, and it's stored in a tree structure. Each HTML element becomes an object in this tree. The `document` object serves as the entry point into the DOM.

2. **Tree Structure**: The DOM is structured like a family tree, with parent, child, and sibling relationships.
   - The root of the tree is the `document` object.
   - The first child of `document` is usually the `html` element.
   - `html` has child elements like `head` and `body`.
   - These elements have their own children, creating a nested structure.

Example DOM Tree

Consider this simple HTML document:

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Page</title>
  </head>
  <body>
    <section>
      <p>This is a paragraph with a <a href="#">link</a>.</p>
    </section>
  </body>
</html>
```

This HTML corresponds to a DOM tree where each element and text is represented as a node. The tree starts with the `document` object, followed by `html`, `head`, `body`, and their respective child elements.

Accessing the DOM with JavaScript

You can interact with the DOM using JavaScript by selecting elements and manipulating them. For example, to select and log the text content of a paragraph:

HTML:

```
<p class="message">Start guessing...</p>
```

JavaScript:

```
console.log(document.querySelector(".message").textContent); // Logs "Start
guessing..."
```

- `document.querySelector('.message')`: Selects the element with the class `message`.
- `.textContent`: Retrieves the text content of the selected element.

DOM is Not Part of JavaScript

It's essential to understand that the DOM and related methods (like `document.querySelector`) are not part of JavaScript itself. JavaScript is defined by the ECMAScript specification, and the DOM is part of the Web APIs provided by browsers.

What are Web APIs?

Web APIs are libraries implemented by browsers that provide additional functionalities to JavaScript. These APIs are automatically available for use in our JavaScript code without any need for importing.

Practical Example: Guess My Number! Game

Let's implement some basic DOM manipulations for our "Guess My Number!" game:

1. **Select Elements**:

```
const messageElement = document.querySelector(".message");
console.log(messageElement.textContent); // Logs "Start guessing..."
```

2. **Change Text Content**:

```
messageElement.textContent = "Correct Number!";
```

3. **Change CSS Styles**:

```
document.querySelector("body").style.backgroundColor = "#60b347";
```

Selecting and Manipulating DOM Elements

Now, we will build on that knowledge by setting the content of elements and exploring further DOM manipulations.

Setting Text Content

To change the text content of an element, we can use the `textContent` property. Here's how to change the text of the element with the class `message`:

HTML:

```
<p class="message">Start guessing...</p>
```

JavaScript:

```
document.querySelector(".message").textContent = "Correct number 🎉";
```

When you save and run this, the text content of the element changes from "Start guessing..." to "Correct number 🎉".

Manipulating More Elements

Next, let's manipulate the content of other elements such as a secret number and a score.

HTML:

```
<div class="number">?</div>
<p>Score: <span class="score">20</span></p>
```

JavaScript:

```
document.querySelector(".number").textContent = 13;
document.querySelector(".score").textContent = 10;
```

This changes the content of the element with the class `number` to 13 and the score to 10.

Working with Input Fields

Input fields are a bit different as they allow users to input data. We can both get and set the value of an input field.

HTML:

```
<input type="number" class="guess" />
```

JavaScript:

```
// Select the input field
const guessInput = document.querySelector(".guess");

// Set the value of the input field
guessInput.value = 23;

// Get the value of the input field and log it to the console
console.log(guessInput.value); // Logs "23"
```

Summary

1. **Selecting Elements**:
    - Use `document.querySelector('.className')` to select elements by class.
    - Use `document.querySelector('#idName')` to select elements by ID.
    - Use `document.querySelector('tagName')` to select elements by tag name.
2. **Setting Text Content**:

```
document.querySelector(".elementClass").textContent = "New Text";
```

3. **Getting and Setting Input Field Values**:

```
const inputField = document.querySelector(".inputClass");
inputField.value = "New Value"; // Set value
console.log(inputField.value); // Get value
```

Practical Example: Guess My Number! Game

Let's implement the above concepts in our "Guess My Number!" game:

HTML:

```
<p class="message">Start guessing...</p>
<div class="number">?</div>
<p>Score: <span class="score">20</span></p>
<input type="number" class="guess" />
```

JavaScript:

```
// Change message text content
document.querySelector(".message").textContent = "Correct number 🎉";

// Change secret number and score
document.querySelector(".number").textContent = 13;
document.querySelector(".score").textContent = 10;

// Set and get input field value
const guessInput = document.querySelector(".guess");
guessInput.value = 23;
console.log(guessInput.value); // Logs "23"
```

Reacting to Button Clicks with Event Listeners

we will make our application react to the Check button click by using an event listener. This will be the first step in making our "Guess My Number!" game interactive.

Adding an Event Listener

An event listener waits for a specific event to happen on an element and then executes a function in response. In our case, we will listen for a click event on the Check button and execute some code when that event occurs.

Step-by-Step Implementation

1. **Select the Check Button**

- Use `querySelector` to select the button with the class `check`.

2. **Add an Event Listener**
   - Use the `addEventListener` method to listen for the click event and execute a function when the event occurs.
3. **Retrieve and Log the Input Value**
   - Inside the event handler function, retrieve the value from the input field with the class `guess` and log it to the console.

Here's the implementation:

HTML:

```
<button class="check">Check!</button>
<input type="number" class="guess" />
<p class="message">Start guessing...</p>
```

JavaScript:

```
// Select the Check button
const checkButton = document.querySelector(".check");

// Add event listener to the Check button
checkButton.addEventListener("click", function () {
  // Get the value from the input field
  const guess = document.querySelector(".guess").value;

  // Log the input value to the console
  console.log(guess);
});
```

When you enter a number in the input field and click the Check button, the value will be logged to the console.

Enhancing the Event Handler

Let's improve the event handler by:

1. **Storing the Input Value in a Variable**
2. **Converting the Input Value to a Number**
3. **Checking if the Input is Empty**
4. **Updating the Message Based on the Input**

Here is the updated code:

JavaScript:

```
// Select the Check button
const checkButton = document.querySelector(".check");

// Add event listener to the Check button
checkButton.addEventListener("click", function () {
  // Get the value from the input field and convert it to a number
  const guess = Number(document.querySelector(".guess").value);

  // Check if the input is empty
  if (!guess) {
    // Update the message if no number is entered
    document.querySelector(".message").textContent = "No number! 🚫";
  } else {
    // Log the input value to the console
    console.log(guess);
  }
});
```

Summary

1. **Selecting Elements**:
   - Use `document.querySelector('.className')` to select elements by class.
2. **Adding Event Listeners**:

```
element.addEventListener("eventType", function () {
  // Code to execute when the event occurs
});
```

3. **Retrieving and Converting Input Values**:

```
const inputValue = Number(document.querySelector(".inputClass").value);
```

4. **Conditional Statements**:

```
if (!inputValue) {
  // Code to execute if input is empty
}
```

Practical Example: Guess My Number! Game

HTML:

```
<button class="check">Check!</button>
<input type="number" class="guess" />
<p class="message">Start guessing...</p>
```

JavaScript:

```
// Select the Check button
const checkButton = document.querySelector(".check");

// Add event listener to the Check button
checkButton.addEventListener("click", function () {
  // Get the value from the input field and convert it to a number
  const guess = Number(document.querySelector(".guess").value);

  // Check if the input is empty
  if (!guess) {
    // Update the message if no number is entered
    document.querySelector(".message").textContent = "No number! 🚫";
  } else {
    // Log the input value to the console
    console.log(guess);
  }
});
```

Great! It looks like we've successfully implemented the core game logic for our guessing game. Let's summarize and clarify the key points and steps involved:

1. **Defining the Secret Number**:
    - We define the secret number outside the button handler to ensure it is only set once when the application starts.
    - We use `Math.random()` to generate a random number between 1 and 20.
2. **Displaying the Secret Number**:
    - For development purposes, we display the secret number in the UI to test our game logic.
3. **Handling User Guesses**:
    - We check if the user input (guess) is equal to, greater than, or less than the secret number.
    - We display appropriate messages for each scenario:
        - If the guess is correct, we display a success message.
        - If the guess is too high or too low, we display corresponding messages.
4. **Tracking the Score**:
    - We maintain a score variable initialized to 20.
    - Each time the user makes an incorrect guess, we decrease the score by 1.
    - We update the score in the DOM after each guess.

5. **Handling Game Over**:
    - We implement logic to handle the game over scenario when the score reaches zero.
    - We display a "You lost the game" message and stop further guesses when the score is zero.

Here's the full code for our current game logic:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Guess My Number!</title>
    <style>
      /* Add your CSS styles here */
    </style>
  </head>
  <body>
    <div class="game">
      <p class="message">Start guessing...</p>
      <p class="number">?</p>
      <input type="number" class="guess" />
      <button class="check">Check!</button>
      <p class="label-score">💯 Score: <span class="score">20</span></p>
    </div>
    <script>
      // Secret number
      const secretNumber = Math.trunc(Math.random() * 20) + 1;
      document.querySelector(".number").textContent = secretNumber;

      // Initial score
      let score = 20;

      // Event listener for the Check button
      document.querySelector(".check").addEventListener("click", function () {
        const guess = Number(document.querySelector(".guess").value);

        // No input
        if (!guess) {
          document.querySelector(".message").textContent = "⛔ No number!";
        }
        // Correct guess
        else if (guess === secretNumber) {
          document.querySelector(".message").textContent = "🎉 Correct number!";
        }
        // Wrong guess
        else if (guess !== secretNumber) {
          if (score > 1) {
            document.querySelector(".message").textContent =
              guess > secretNumber ? "📈 Too high!" : "📉 Too low!";
            score--;
            document.querySelector(".score").textContent = score;
          } else {
```

```
            document.querySelector(".message").textContent =
              "💥 You lost the game!";
            document.querySelector(".score").textContent = 0;
          }
        }
      });
    </script>
  </body>
</html>
```

Manipulating CSS Styles in JavaScript

Goal

To change the background color of the page to green and increase the width of the number display when the player wins the game.

Steps

1. **Select the Element:**
   - Use `document.querySelector` to select the desired element.

```
document.querySelector("body"); // Selects the body element
document.querySelector(".number"); // Selects the element with the class
'number'
```

2. **Change CSS Styles:**
   - Access the `style` property of the selected element and modify the CSS properties using camelCase notation for properties with hyphens.

```
// Change background color of the body
document.querySelector("body").style.backgroundColor = "#60b347";

// Change width of the number element
document.querySelector(".number").style.width = "30rem";
```

Detailed Explanation

- **Selecting Elements:**
  - `document.querySelector('body')` selects the entire body element.
  - `document.querySelector('.number')` selects the element with the class `number`.
- **Modifying Styles:**

- The `style` property allows you to access and modify the inline styles of an element.
- CSS properties that contain hyphens, like `background-color`, must be written in camelCase in JavaScript, e.g., `backgroundColor`.
- When setting styles, the value must be a string, including units if necessary (e.g., '30rem').

Example Code

Here's the complete code for changing the background color and width when the player wins the game:

```
if (guess === secretNumber) {
  // Player wins
  document.querySelector("body").style.backgroundColor = "#60b347";
  document.querySelector(".number").style.width = "30rem";
}
```

Additional Notes

- **Inline Styles:**
  - The styles set using JavaScript are applied as inline styles, which means they are directly written into the HTML element's `style` attribute.
  - This does not change the external CSS file.
- **Comments for Clarity:**
  - Adding comments in your code helps make it more understandable. For example:

```
// When player wins
if (guess === secretNumber) {
  // Change background color to green
  document.querySelector("body").style.backgroundColor = "#60b347";
  // Increase width of the number display
  document.querySelector(".number").style.width = "30rem";
}
```

Practical Application

By following these steps, you can manipulate various CSS properties dynamically in your web applications, enhancing user interactions and providing visual feedback based on user actions. Great! Here's a step-by-step breakdown to implement the functionality of the "Again" button to reset the game:

Resetting the Game with the "Again" Button

Goal

To reset the game so that the player can play again without reloading the page.

Steps

1. **Fix Initial State Display:**
    - Move the display of the secret number to the win condition.
2. **Implement the "Again" Button Functionality:**
    - Select the "Again" button element.
    - Attach a click event handler to the "Again" button.
    - In the event handler function, restore the initial values and conditions.

Detailed Implementation

1. **Fix Initial State Display:**

   Move the display of the secret number to the win condition.

```
if (guess === secretNumber) {
  // Player wins
  document.querySelector(".number").textContent = secretNumber; // Display
the secret number
  document.querySelector("body").style.backgroundColor = "#60b347";
  document.querySelector(".number").style.width = "30rem";
}
```

2. **Implement the "Again" Button Functionality:**
    - **Select the "Again" Button Element:**

```
const againButton = document.querySelector(".again");
```

    - **Attach Click Event Handler:**

```
againButton.addEventListener("click", function () {
  // Restore initial values of the score and secret number
  score = 20;
  secretNumber = Math.trunc(Math.random() * 20) + 1;

  // Restore initial conditions of the message, number, score, and
guess input field
  document.querySelector(".message").textContent = "Start guessing...";
  document.querySelector(".score").textContent = score;
  document.querySelector(".number").textContent = "?";
  document.querySelector(".guess").value = "";

  // Restore original background color and width of the number
  document.querySelector("body").style.backgroundColor = "#222";
```

```
    document.querySelector(".number").style.width = "15rem";
  });
```

Example Code

Here's the complete code incorporating all the steps:

```
// Winning condition
if (guess === secretNumber) {
  document.querySelector(".message").textContent = "Correct Number!";
  document.querySelector(".number").textContent = secretNumber;
  document.querySelector("body").style.backgroundColor = "#60b347";
  document.querySelector(".number").style.width = "30rem";
}

// Reset functionality
const againButton = document.querySelector(".again");
againButton.addEventListener("click", function () {
  // Restore initial values of the score and secret number
  score = 20;
  secretNumber = Math.trunc(Math.random() * 20) + 1;

  // Restore initial conditions of the message, number, score, and guess input
field
  document.querySelector(".message").textContent = "Start guessing...";
  document.querySelector(".score").textContent = score;
  document.querySelector(".number").textContent = "?";
  document.querySelector(".guess").value = "";

  // Restore original background color and width of the number
  document.querySelector("body").style.backgroundColor = "#222";
  document.querySelector(".number").style.width = "15rem";
});
```

Explanation

- **Selecting the "Again" Button:**
  - `document.querySelector('.again')` selects the button with the class `again`.
- **Attaching Click Event Handler:**
  - `addEventListener('click', function() { ... })` attaches a click event handler to the button.
  - The function inside the event handler restores all initial values and conditions, effectively resetting the game.
- **Restoring Initial Values and Conditions:**
  - **Score and Secret Number:**

        - Reset `score` to 20.
        - Generate a new `secretNumber`.
    - **UI Elements:**
        - Reset the message to "Start guessing...".
        - Reset the score display to the initial score.
        - Reset the number display to "?".
        - Clear the guess input field.
    - **Styles:** - Reset the background color to the original color. - Reset the width of the number display to the original width. To implement the high score functionality in your game, follow these detailed steps:

High Score Functionality

Steps

1. **Declare a High Score Variable:**
    - Initialize a `highScore` variable to store the highest score achieved.
2. **Update the High Score:**
    - Check if the current score is greater than the high score when the player wins.
    - If it is, update the high score variable and display it.

Implementation

1. **Declare a High Score Variable:**

```
let highScore = 0;
```

2. **Update the High Score When Player Wins:**
    - Check if the current score is greater than the high score.
    - If it is, update the high score and display it.

```
if (guess === secretNumber) {
  document.querySelector(".message").textContent = "Correct Number!";
  document.querySelector(".number").textContent = secretNumber;
  document.querySelector("body").style.backgroundColor = "#60b347";
  document.querySelector(".number").style.width = "30rem";

  if (score > highScore) {
    highScore = score;
    document.querySelector(".highscore").textContent = highScore;
  }
}
```

3. **Display the High Score in the HTML:**
    - Ensure you have an element in your HTML with the class `highscore` to display the high score.

Example Code

Here's the complete code incorporating all the steps:

```
// Variables
let secretNumber = Math.trunc(Math.random() * 20) + 1;
let score = 20;
let highScore = 0;

// Check guess
document.querySelector(".check").addEventListener("click", function () {
  const guess = Number(document.querySelector(".guess").value);

  // When there is no input
  if (!guess) {
    document.querySelector(".message").textContent = "No number!";

    // When player wins
  } else if (guess === secretNumber) {
    document.querySelector(".message").textContent = "Correct Number!";
    document.querySelector(".number").textContent = secretNumber;
    document.querySelector("body").style.backgroundColor = "#60b347";
    document.querySelector(".number").style.width = "30rem";

    if (score > highScore) {
      highScore = score;
      document.querySelector(".highscore").textContent = highScore;
    }

    // When guess is wrong
  } else if (guess !== secretNumber) {
    if (score > 1) {
      document.querySelector(".message").textContent =
        guess > secretNumber ? "Too high!" : "Too low!";
      score--;
      document.querySelector(".score").textContent = score;
    } else {
      document.querySelector(".message").textContent = "You lost the game!";
      document.querySelector(".score").textContent = 0;
    }
  }
});

// Again button
document.querySelector(".again").addEventListener("click", function () {
  score = 20;
  secretNumber = Math.trunc(Math.random() * 20) + 1;
```

```
  document.querySelector(".message").textContent = "Start guessing...";
  document.querySelector(".score").textContent = score;
  document.querySelector(".number").textContent = "?";
  document.querySelector(".guess").value = "";

  document.querySelector("body").style.backgroundColor = "#222";
  document.querySelector(".number").style.width = "15rem";
});
```

Explanation

- **High Score Variable:**
  - `let highScore = 0;` initializes the high score to zero.
- **Updating the High Score:**
  - Inside the winning condition (`if (guess === secretNumber)`), we check if the current score is greater than the high score.
  - If it is, we update the high score and set the text content of the `.highscore` element to the new high score.
- **HTML Element:**
  - Ensure your HTML has an element with the class `highscore` to display the high score, such as:

    ```
    <p>🥇 Highscore: <span class="highscore">0</span></p>
    ```

    To finish this project, let's learn about refactoring to eliminate duplicate code and make our codebase cleaner and more maintainable. The main focus will be on the repeated code in the sections where the guess is either too high or too low.

Refactoring Steps

1. **Identify Duplicate Code**: The code for the conditions where the guess is too high and too low is almost identical. The only difference is the message displayed to the user.
2. **Unify Conditions**: Instead of having separate blocks for "too high" and "too low", we can combine them into a single block that handles any incorrect guess.
3. **Use a Ternary Operator**: This will help us determine the specific message to display based on whether the guess is higher or lower than the secret number.
4. **Create a Function**: To avoid repetitive code for setting messages, we can create a function called `displayMessage`.

Step-by-Step Implementation

1. Unify Conditions

We start by replacing the separate `if` blocks for high and low guesses with a single block that handles both cases.

```
else if (guess !== secretNumber) {
    // If the guess is incorrect
    if (score > 1) {
        document.querySelector('.message').textContent = guess > secretNumber ?
'Too high!' : 'Too low!';
        score--;
        document.querySelector('.score').textContent = score;
    } else {
        document.querySelector('.message').textContent = 'You lost the game!';
        document.querySelector('.score').textContent = 0;
    }
}
```

2. Create the `displayMessage` Function

We create a function to handle setting the message text content, reducing repetition.

```
function displayMessage(message) {
  document.querySelector(".message").textContent = message;
}
```

3. Replace Repetitive Code with the Function

We update our game logic to use the new `displayMessage` function.

```
else if (guess !== secretNumber) {
    if (score > 1) {
        displayMessage(guess > secretNumber ? 'Too high!' : 'Too low!');
        score--;
        document.querySelector('.score').textContent = score;
    } else {
        displayMessage('You lost the game!');
        document.querySelector('.score').textContent = 0;
    }
}
```

Now, all occurrences of `document.querySelector('.message').textContent = ...` are replaced with calls to `displayMessage`.

Final Code

Here is the final version of the updated game logic after refactoring:

```
let score = 20;
let highscore = 0;
let secretNumber = Math.trunc(Math.random() * 20) + 1;

document.querySelector(".check").addEventListener("click", function () {
  const guess = Number(document.querySelector(".guess").value);

  if (!guess) {
    displayMessage("No number!");
  } else if (guess === secretNumber) {
    displayMessage("Correct Number!");
    document.querySelector(".number").textContent = secretNumber;

    document.querySelector("body").style.backgroundColor = "#60b347";
    document.querySelector(".number").style.width = "30rem";

    if (score > highscore) {
      highscore = score;
      document.querySelector(".highscore").textContent = highscore;
    }
  } else if (guess !== secretNumber) {
    if (score > 1) {
      displayMessage(guess > secretNumber ? "Too high!" : "Too low!");
      score--;
      document.querySelector(".score").textContent = score;
    } else {
      displayMessage("You lost the game!");
      document.querySelector(".score").textContent = 0;
    }
  }
});

document.querySelector(".again").addEventListener("click", function () {
  score = 20;
  secretNumber = Math.trunc(Math.random() * 20) + 1;

  displayMessage("Start guessing...");
  document.querySelector(".score").textContent = score;
  document.querySelector(".number").textContent = "?";
  document.querySelector(".guess").value = "";

  document.querySelector("body").style.backgroundColor = "#222";
  document.querySelector(".number").style.width = "15rem";
});

function displayMessage(message) {
  document.querySelector(".message").textContent = message;
}
```

Summary

We improved our code by:

1. Combining similar conditional blocks to eliminate redundancy.
2. Creating a reusable function (`displayMessage`) to handle setting the message text content.
3. Refactoring the code to follow the DRY (Don't Repeat Yourself) principle, making it cleaner and easier to maintain.

Modal Window Project Overview

In this project, we will create a simple yet essential UI component—a modal window. The modal window is a common feature in web applications, typically used to display information or prompt user interactions without leaving the current page. This modal will open when any of the buttons on the page are clicked, and it can be closed in three ways:

1. By clicking the close button within the modal.
2. By clicking outside the modal on the overlay.
3. By pressing the "Esc" key on the keyboard.

Key Concepts Covered:

- **Class Manipulation**: This project will focus on manipulating classes, which is crucial for dynamically changing the appearance and behavior of elements on a webpage.
- **Element Selection**: Efficiently selecting and storing elements in variables to avoid redundant selections.

Steps to Implement the Modal Window

1. **Setting Up the Project**:
    - We start by creating a new folder for the project and open it in VS Code.
    - The project includes three files: `index.html`, `style.css`, and `script.js`.
    - `index.html` contains the structure of the modal, overlay, and buttons.
    - `style.css` defines the styling, including a `hidden` class that initially hides the modal.
    - `script.js` will handle the functionality, including showing and hiding the modal.
2. **Starting the Live Server**:
    - Use VS Code's terminal to start the Live Server, which automatically reloads the webpage on code changes.
3. **Element Selection**:
    - We begin by selecting all the necessary elements from the DOM using `document.querySelector` and `document.querySelectorAll`.
    - Selected elements include the modal, overlay, close button, and the show modal buttons.
    - **Example**:

```
const modal = document.querySelector(".modal");
const overlay = document.querySelector(".overlay");
const btnCloseModal = document.querySelector(".close-modal");
const btnsOpenModal = document.querySelectorAll(".show-modal");
```

4. **Handling Multiple Elements**:
    - The `querySelectorAll` method is used to select multiple elements (in this case, the three buttons that open the modal). This method returns a NodeList, which we can loop through using a `for` loop.
    - Example: Logging the text content of each button to the console:

```
for (let i = 0; i < btnsOpenModal.length; i++) {
  console.log(btnsOpenModal[i].textContent);
}
```

Conclusion:

In the next steps, we will add functionality to make the modal window interactive. This involves adding event listeners to open and close the modal and implementing the different ways to close it.

Manipulating Classes with JavaScript

In this session, we'll learn how to manipulate classes in JavaScript, which is an essential skill for dynamically changing the appearance of elements on a webpage. Here's what we'll cover:

1. **Attaching Event Handlers to Multiple Buttons**
2. **Showing and Hiding a Modal**
3. **Adding and Removing Classes**
4. **Refactoring Code for Reusability**

---

1. Attaching Event Handlers to Multiple Buttons

To start, we need to respond to a click event on each button. Here's how we can do it:

- **Selecting Buttons**: We use a `for` loop to iterate over a list of buttons (`btnsOpenModal`). Each button element is accessed in the loop.
- **Adding Event Listeners**: We attach an event listener to each button using the `addEventListener` method, which listens for the 'click' event. The code to execute on click is placed inside an anonymous function.

Example:

```
btnsOpenModal[i].addEventListener("click", function () {
  console.log("Button clicked");
});
```

2. Showing and Hiding a Modal

Next, we want to display the modal when a button is clicked.

- **Manipulating the Class List**: The modal is hidden by default using a `hidden` class. To show it, we remove the `hidden` class using the `classList.remove` method.

Example:

```
modal.classList.remove("hidden");
overlay.classList.remove("hidden");
```

To hide the modal again when a close button or the overlay is clicked, we add the `hidden` class back.

Example:

```
modal.classList.add("hidden");
overlay.classList.add("hidden");
```

3. Adding and Removing Classes

Using classes to control styles is preferable over directly manipulating individual style properties in JavaScript. It allows for more organized, scalable code. For example:

- **Direct Style Manipulation**:

  ```
  modal.style.display = "block";
  ```

- **Class-Based Manipulation**:

  ```
  modal.classList.remove("hidden");
  ```

The latter is preferred because it aggregates multiple styles into a single class.

4. Refactoring Code for Reusability

To avoid repeating code, we refactor by creating named functions for the operations:

- **Opening the Modal**:

  ```
  const openModal = function () {
    modal.classList.remove("hidden");
    overlay.classList.remove("hidden");
  };
  ```

- **Closing the Modal**:

```
const closeModal = function () {
  modal.classList.add("hidden");
  overlay.classList.add("hidden");
};
```

These functions can then be reused across multiple event listeners, making the code cleaner and more maintainable.

Key Takeaways

- **Event Listeners**: Attach event handlers to elements using `addEventListener`.
- **Class Manipulation**: Use `classList.add`, `classList.remove`, and `classList.contains` to manipulate classes.
- **Refactoring**: Refactor repetitive code into reusable functions for better code organization and readability.

Handling Key Press Events in JavaScript

Introduction

In this lesson, we learned how to handle keyboard events in JavaScript, specifically focusing on closing a modal when the "Escape" key is pressed. We explored how to use the `addEventListener` method to listen for global events like key presses, how to access the event object, and how to conditionally close the modal based on the key pressed.

Listening for Keyboard Events

- **Global Events**: Keyboard events are considered global events, meaning they do not happen on a specific element but can occur anywhere on the page.
- **Event Listener**: To listen for these events, we use `document.addEventListener('keydown', function)`. The `'keydown'` event is triggered as soon as a key is pressed down.

```
document.addEventListener("keydown", function () {
  console.log("A key was pressed");
});
```

Accessing the Event Object

- **Event Object**: When an event like a key press occurs, JavaScript generates an event object containing information about the event. We can access this object in our event handler by passing a parameter to the function (commonly named `e` or `event`).

```
document.addEventListener("keydown", function (e) {
  console.log(e.key); // Logs the key that was pressed
```

```
});
```

Implementing the Escape Key Functionality

- **Conditionally Closing the Modal**: We check if the pressed key is the "Escape" key by evaluating `e.key`. If it's "Escape" and the modal is visible (does not contain the `hidden` class), we close the modal by calling the `closeModal` function.

```
document.addEventListener("keydown", function (e) {
  if (e.key === "Escape" && !modal.classList.contains("hidden")) {
    closeModal();
  }
});
```

Code Optimization

- **Combining Conditions**: The two conditions (`e.key === 'Escape'` and `!modal.classList.contains('hidden')`) can be combined into a single `if` statement to make the code more concise and readable.

```
document.addEventListener("keydown", function (e) {
  if (e.key === "Escape" && !modal.classList.contains("hidden")) {
    closeModal();
  }
});
```

Final Testing

- **Testing**: After implementing the functionality, testing involved opening the modal and pressing various keys to ensure only the "Escape" key closed the modal.

Conclusion

This lesson demonstrated how to handle key press events, specifically for closing a modal with the "Escape" key.

Section 12: Numbers , Dates, INTL , Timers

Understanding Numbers in JavaScript

Internal Representation of Numbers

- **Single Data Type:** In JavaScript, all numbers are represented as floating-point numbers, even if they are written as integers (e.g., `23` is the same as `23.0`).

- **Binary Format:** Numbers are stored in a 64-bit binary (base 2) format, which can lead to imprecise representations of some fractions, especially those that are easy to represent in base 10. For example, `0.1 + 0.2` results in `0.30000000000000004` due to these limitations.
- **Implications:** This imprecision means JavaScript may not be suitable for highly precise calculations, especially in scientific or financial contexts.

Converting Values to Numbers

- **Using `Number()` Function:** You can convert strings to numbers using `Number('23')`, which will return `23`.
- **Using `+` Operator:** A shorthand method is to use the `+` operator, such as `+'23'`, which also converts the string to a number. This method is cleaner and more concise.

Parsing Numbers from Strings

- `parseInt()` **Function:**
  - Converts a string to an integer by parsing it from the start of the string.
  - If the string starts with non-numeric characters, it returns `NaN`.
  - Example: `Number.parseInt('30px')` returns `30`.
  - **Base Argument:** `parseInt()` accepts a second argument to specify the base of the numeral system (e.g., base 10). `Number.parseInt('1010', 2)` would interpret `'1010'` as a binary number and return `10`.
- `parseFloat()` **Function:**
  - Converts a string to a floating-point number, including decimals.
  - Example: `Number.parseFloat('2.5rem')` returns `2.5`.
  - **Whitespace Handling:** It ignores leading and trailing whitespaces.
- **Global vs. Namespace:** `parseInt()` and `parseFloat()` can be called as global functions or through the `Number` object. The latter is more modern and preferred.

Checking if Values Are Numbers

- `isNaN()` **Function:**
  - Checks if a value is `NaN` (Not-a-Number).
  - Example: `Number.isNaN(Number('string'))` returns `true` because converting a non-numeric string to a number results in `NaN`.
  - **Limitations:** It can be misleading in some cases (e.g., `Number.isNaN(20)` returns `false` even if `20` is a string).
- `isFinite()` **Function:**
  - A better method for checking if a value is a number.
  - Example: `Number.isFinite(20)` returns `true`, while `Number.isFinite('20')` returns `false`.
  - This function ensures that the value is a real, finite number, excluding `NaN` and `Infinity`.
- `isInteger()` **Function:**
  - Checks if a value is an integer.
  - Example: `Number.isInteger(23)` returns `true`, while `Number.isInteger(23.5)` returns `false`.

Practical Application

- **String to Number Conversion in Projects:** When working on projects like the Bankist app, using `+` to convert strings to numbers can make your code cleaner.
- **Parsing and Validating Input:** Use `parseFloat()` when reading numeric values from strings (e.g., CSS units). For checking whether a value is a valid number, prefer `isFinite()` over `isNaN()`.

Key Takeaways

- **Internal Representation:** Be aware of the limitations of floating-point numbers in JavaScript.
- **Conversions:** Use `+` for quick conversions and `parseInt()`/`parseFloat()` for parsing numbers from strings.
- **Validation:** Use `isFinite()` to accurately check if a value is a valid number.

Mathematical Operations and Rounding in JavaScript

Mathematical Operations

1. **Square Root:**
   - Use `Math.sqrt(number)` to get the square root.
   - Example: `Math.sqrt(9)` returns `3`.

   Alternatively, you can use the exponentiation operator:

   ```
   const squareRoot = 9 ** (1 / 2); // 3
   ```

   For cubic roots:

   ```
   const cubicRoot = 8 ** (1 / 3); // 2
   ```

2. **Maximum and Minimum Values:**
   - Use `Math.max(value1, value2, ...)` to get the maximum value.

     ```
     Math.max(10, 20, 5); // 20
     ```

   - Use `Math.min(value1, value2, ...)` to get the minimum value.

     ```
     Math.min(10, 20, 5); // 5
     ```

   Note: `Math.max` and `Math.min` do type coercion but not parsing. For example, `Math.max("10px", 20)` will return `20` as `"10px"` is not a number.

3. **Math Constants:**

   - Use `Math.PI` for the value of π (pi), useful for calculations involving circles.

```
const radius = 10;
const area = Math.PI * radius ** 2; // Area of a circle
```

4. **Random Numbers:**

   - `Math.random()` generates a random number between `0` (inclusive) and `1` (exclusive).

```
Math.random(); // e.g., 0.234
```

   - To get a random integer between `min` and `max`, use:

```
function randomInt(min, max) {
  return Math.floor(Math.random() * (max - min + 1)) + min;
}

randomInt(10, 20); // e.g., 15
```

Rounding Numbers

1. **Integer Rounding:**

   - `Math.trunc(number)` removes the decimal part and returns the integer.

```
Math.trunc(23.9); // 23
```

   - `Math.round(number)` rounds to the nearest integer.

```
Math.round(23.5); // 24
```

   - `Math.ceil(number)` rounds up to the nearest integer.

```
Math.ceil(23.1); // 24
```

   - `Math.floor(number)` rounds down to the nearest integer.

```
```

```
Math.floor(23.9); // 23
```

Comparison:

- `Math.trunc` and `Math.floor` are similar for positive numbers but differ for negative numbers. `Math.floor` rounds towards negative infinity, while `Math.trunc` simply cuts off the decimal part.

2. **Decimal Rounding:**

   - Use `.toFixed(digits)` to round to a specified number of decimal places.

   ```
   const num = 2.34567;
   num.toFixed(2); // "2.35" (returns a string)
   ```

   - Convert the string result back to a number:

   ```
   parseFloat(num.toFixed(2)); // 2.35
   ```

   Note: `toFixed` returns a string with the specified number of decimal places, so converting it back to a number requires `parseFloat` or the unary plus operator (`+`).

Additional Notes

- **Boxing:** Primitive types in JavaScript (like numbers) don't have methods. JavaScript uses "boxing" to convert primitives into objects temporarily to call methods on them. After the method call, it converts them back to primitives. This process allows methods like `toFixed` to be used on numbers.

Mathematical Operations and Rounding Numbers

Mathematical Operations Recap

- **Square Root:**
  - Use `Math.sqrt(number)` to calculate the square root.
  - Alternatively, you can use exponentiation: `number ** (1/2)` for square root, `number ** (1/3)` for cubic root.
- **Maximum and Minimum Values:**
  - `Math.max(value1, value2, ...)` returns the maximum value.
  - `Math.min(value1, value2, ...)` returns the minimum value.
- **Mathematical Constants:**
  - `Math.PI` is a constant representing the value of π, useful for calculations involving circles.
- **Random Numbers:**

- `Math.random()` generates a random number between 0 and 1.
- You can generate a random integer between two values by using a custom function.

Rounding Numbers

- **Truncation:**
  - `Math.trunc(number)` removes the decimal part of the number.
- **Rounding to Nearest Integer:**
  - `Math.round(number)` rounds the number to the nearest integer.
- **Ceiling and Floor:**
  - `Math.ceil(number)` always rounds up to the nearest integer.
  - `Math.floor(number)` always rounds down to the nearest integer.
- **Handling Decimals:**
  - Use `number.toFixed(decimalPlaces)` to round a number to a specific number of decimal places. Note that this returns a string.

Remainder Operator (`%`)

Definition

- The remainder operator returns the remainder of the division of two numbers.

Use Cases

- **Checking Even and Odd Numbers:**
  - A number is even if `number % 2 === 0`.
  - A number is odd if `number % 2 !== 0`.
- **Divisibility:**
  - The remainder operator can be used to check if a number is divisible by another number. If `number % divisor === 0`, the number is divisible by the divisor.
- **Styling Rows in a Table:**
  - You can use the remainder operator to apply styles to every nth row in a table (e.g., every second or third row).

Example: Remainder Operator in Action

```
// Function to check if a number is even
const isEven = (n) => n % 2 === 0;

console.log(isEven(8)); // true
```

```
console.log(isEven(23)); // false

// Styling every second and third row in a table
const rows = Array.from(document.querySelectorAll(".row"));
rows.forEach((row, index) => {
  if (index % 2 === 0) {
    row.style.backgroundColor = "orangeRed";
  }
  if (index % 3 === 0) {
    row.style.backgroundColor = "blue";
  }
});
```

Key Takeaways

- The remainder operator is a versatile tool in mathematical operations and can be used for tasks such as checking divisibility, alternating styles, and more.

Numeric Separators in JavaScript

Introduction to Numeric Separators

- **Feature Introduction:**
  - Numeric separators, introduced in 2021, make it easier to read and understand large numbers in JavaScript.
  - They are represented by underscores (_) that can be placed within numbers to visually separate digits.

Why Use Numeric Separators?

- **Enhanced Readability:**
  - Large numbers can be difficult to read when written out normally, especially when they contain many zeros.
  - For example, the number `287460000000` is hard to parse at a glance.
  - By using numeric separators, you can write the number as `287_460_000_000`, making it immediately clear that this is 287 billion.

How JavaScript Handles Numeric Separators

- **JavaScript Interpretation:**
  - When numeric separators are used, the JavaScript engine ignores them and treats the number as if the underscores are not present.
  - This means that `287_460_000_000` is treated the same as `287460000000` by JavaScript.

- **Examples:**

```
const diameter = 287_460_000_000;
console.log(diameter); // Output: 287460000000
```

```
const priceInCents = 34_599;
console.log(priceInCents); // Output: 34599
```

Use Cases for Numeric Separators

- **Meaningful Separations:**
  - Numeric separators can be used to differentiate parts of a number, such as separating thousands, millions, or different currency denominations.
  - **Examples:**
    - **Thousands Separator:**

```
const transferFee = 1_500; // Interpreted as 1500
```

    - **Cents Representation:**

```
const priceInCents = 34_599; // Interpreted as 34599 cents
```

Restrictions on Numeric Separators

- **Placement Rules:**
  - Underscores can only be placed between digits.
  - **Illegal placements include:**
    - **At the beginning of a number:** `_1000` (Error)
    - **At the end of a number:** `1000_` (Error)
    - **Adjacent to a decimal point:** `3_.1415` or `3._1415` (Error)
    - **Consecutive underscores:** `100__000` (Error)

```
const PI = 3.14_15; // Valid
const invalidPI = 3._1415; // Error
```

Conversion Issues with Numeric Separators

- **String to Number Conversion:**
  - If you try to convert a string containing underscores to a number using `Number()` or `parseInt()`, it won't work as expected.

- **Example:**

```
const str = "230_000";
console.log(Number(str)); // Output: NaN
console.log(parseInt(str)); // Output: 230
```

  - `Number()` returns `NaN` because it doesn't recognize the string as a valid number.
  - `parseInt()` only converts up to the underscore, ignoring the rest.

- **Best Practice:**
  - Use numeric separators only when writing actual numbers in your code, not when handling numbers as strings.
  - Avoid using underscores in strings, especially when dealing with APIs or data storage, as it may lead to unexpected errors.

Key Takeaways

- Numeric separators improve the readability of large numbers in JavaScript by allowing you to insert underscores as visual separators.
- The JavaScript engine ignores these separators, ensuring the number is processed correctly.
- Be mindful of where you place numeric separators and avoid using them in strings that represent numbers to prevent parsing issues.

BigInt in JavaScript

Introduction to BigInt

- **What is BigInt?**
  - BigInt is a special type of integer introduced in JavaScript in 2020.
  - It allows for the representation of numbers larger than the `Number.MAX_SAFE_INTEGER`, which is `2^53 - 1`.
  - BigInt can store integers of arbitrary size, making it possible to work with extremely large numbers that regular JavaScript numbers (which are limited to 64-bit) cannot accurately represent.

Understanding Number Representation in JavaScript

- **Internal Representation of Numbers:**
  - JavaScript numbers are stored internally using 64 bits, where only 53 bits are used to store the digits.
  - The remaining bits are used to store the sign and the position of the decimal point.
  - The maximum safe integer in JavaScript, which can be represented without loss of precision, is `2^53 - 1`.

- **Max Safe Integer Example:**

```
const maxSafeInteger = Number.MAX_SAFE_INTEGER;
console.log(maxSafeInteger); // Output: 9007199254740991
```

- **Unsafe Numbers:**
  - Any number larger than `Number.MAX_SAFE_INTEGER` is considered unsafe, as JavaScript cannot guarantee precision with these numbers.
  - Adding 1 to `Number.MAX_SAFE_INTEGER` does not yield the expected result:

```
console.log(9007199254740991 + 1); // Output: 9007199254740992
console.log(9007199254740991 + 2); // Output: 9007199254740992
```

  - As seen, adding 1 and adding 2 both give the same result, showing the loss of precision.

Creating BigInt

- **Using the `n` Suffix:**
  - A BigInt can be created by appending an `n` to the end of an integer literal:

```
const bigIntNumber = 1234567890123456789012345678901234567890n;
console.log(bigIntNumber); // Output:
1234567890123456789012345678901234567890n
```

- **Using the `BigInt` Function:**
  - You can also create a BigInt using the `BigInt()` function:

```
const bigIntFromFunction =
BigInt(123456789012345678901234567890123456789O);
console.log(bigIntFromFunction); // Output:
1234567890123456789012345678901234567890n
```

  - Note: Converting large numbers using the `BigInt()` function may result in loss of precision if the number exceeds `Number.MAX_SAFE_INTEGER`.

Operations with BigInt

- **Basic Arithmetic Operations:**
  - You can perform basic arithmetic operations with BigInt just like regular numbers:

```
const result = 10000n + 20000n;
console.log(result); // Output: 30000n
```

- **Multiplication Example:**

```
const bigProduct = 10n * 123456789012345678901234567890n;
console.log(bigProduct); // Output: 1234567890123456789012345678900n
```

- **Incompatibility with Regular Numbers:**
  - Mixing BigInt and regular numbers in arithmetic operations will result in an error:

```
const huge = 10000000000000000000000000000000000000n;
const num = 23;
console.log(huge * num); // Error: Cannot mix BigInt and other types
```

  - To resolve this, you must convert the regular number to BigInt:

```
console.log(huge * BigInt(num)); // Output:
230000000000000000000000000000000000000n
```

- **Exceptions:**
  - **Comparison Operators:**
    - BigInt can be compared with regular numbers using comparison operators:

```
console.log(20n > 15); // Output: true
console.log(20n === 20); // Output: false (different types)
console.log(20n == 20); // Output: true (type coercion)
```

  - **String Concatenation:**
    - BigInt can be concatenated with strings:

```
const bigStr = huge + " is really big";
console.log(bigStr); // Output:
"10000000000000000000000000000000000000 is really big"
```

Limitations of BigInt

- **No Support for Math Operations:**
  - BigInt cannot be used with built-in `Math` functions like `Math.sqrt()`:

```
console.log(Math.sqrt(16n)); // Error: Cannot convert a BigInt value to
a number
```

- **Division with BigInt:**
  - Division with BigInt returns the integer result, truncating any decimal part:

```
console.log(10n / 3n); // Output: 3n
console.log(11n / 3n); // Output: 3n
```

Practical Use of BigInt

- **Use Cases:**
  - BigInt is particularly useful in scenarios requiring extremely large numbers, such as working with database IDs, cryptographic applications, or interacting with systems that use 64-bit integers.
- **Conclusion:**
  - While BigInt may not be used frequently in everyday JavaScript programming, it's an essential tool when dealing with large integers beyond the safe range of regular numbers. Understanding how to use BigInt effectively can help prevent precision-related issues in such cases.

Dates and Times in JavaScript

1. Introduction

- Dates and times are essential in real-world applications, and JavaScript provides several ways to handle them.
- Working with dates can be messy and confusing, but understanding the fundamentals will help manage them effectively.

2. Creating Dates

- There are four main ways to create dates using the `Date` constructor in JavaScript.

1. Using `new Date()`

- Creates a date object representing the current date and time.

```
const now = new Date();
console.log(now); // Outputs the current date and time.
```

2. Parsing a Date String

- JavaScript can parse date strings and create a date object.

```
const dateFromString = new Date("August 2, 2020");
console.log(dateFromString); // Outputs: Sun Aug 02 2020...
```

- Custom date strings are parsed, but this can be unreliable unless generated by JavaScript itself.

3. Using Year, Month, Day, etc.

- You can pass year, month, day, hour, minute, and second into the constructor.

```
const futureDate = new Date(2037, 10, 19, 15, 23, 5);
console.log(futureDate); // Outputs: Thu Nov 19 2037 15:23:05...
```

- Note: Months are zero-based (0 = January, 11 = December).

4. Using Timestamps

- Timestamps represent milliseconds passed since January 1, 1970 (Unix epoch).

```
const unixEpoch = new Date(0);
console.log(unixEpoch); // Outputs: Thu Jan 01 1970...

const threeDaysLater = new Date(3 * 24 * 60 * 60 * 1000);
console.log(threeDaysLater); // Outputs: Sun Jan 04 1970...
```

3. Working with Dates

- Date objects have various methods for getting and setting different components.

Getting Date Components

- `getFullYear()`: Returns the year.
- `getMonth()`: Returns the month (0-based).
- `getDate()`: Returns the day of the month.
- `getDay()`: Returns the day of the week (0 = Sunday).
- `getHours()`, `getMinutes()`, `getSeconds()`: Returns the respective time components.

```
const future = new Date(2037, 10, 19, 15, 23);
console.log(future.getFullYear()); // 2037
console.log(future.getMonth()); // 10 (November)
console.log(future.getDate()); // 19
console.log(future.getDay()); // 4 (Thursday)
console.log(future.getHours()); // 15
console.log(future.getMinutes()); // 23
```

Converting Dates to Strings

- `toISOString()`: Converts the date to a standardized ISO string format.

```
console.log(future.toISOString()); // Outputs ISO string of the date.
```

Getting Timestamps

- `getTime()`: Returns the timestamp (milliseconds since the Unix epoch).

```
console.log(future.getTime()); // Outputs the timestamp.
```

- `Date.now()`: Returns the current timestamp without creating a `Date` object.

```
console.log(Date.now()); // Outputs the current timestamp.
```

Setting Date Components

- `setFullYear()`, `setMonth()`, `setDate()`, etc., allow modifying date components.

```
future.setFullYear(2040);
console.log(future); // Date updated to November 19, 2040.
```

Auto-Correction

- JavaScript auto-corrects dates when invalid dates are provided.

```
const incorrectDate = new Date(2037, 10, 31); // November 31, 2037
console.log(incorrectDate); // Auto-corrected to December 1, 2037.
```

4. Conclusion

- Dates in JavaScript can be managed using various methods to create, get, and set components.
- Understanding these basics is crucial for working effectively with dates in JavaScript applications.
- The next step is to apply this knowledge to real-world applications, such as adding dates to a banking application.

Lecture Notes: Implementing Dates in the Application

---

1. Displaying the Current Date on the Interface

- **Objective**: Display the current date on the user interface in a specific format (day/month/year) common in Europe, particularly Portugal.

- **Steps**:
  - Create the current date using `new Date()`.
  - Extract the day, month, and year using `getDate()`, `getMonth()`, and `getFullYear()` methods respectively.
  - Display the formatted date in the UI.

```
const now = new Date();
const day = `${now.getDate()}`.padStart(2, "0");
const month = `${now.getMonth() + 1}`.padStart(2, "0"); // Months are zero-based
const year = now.getFullYear();
const hour = `${now.getHours()}`.padStart(2, "0");
const min = `${now.getMinutes()}`.padStart(2, "0");

labelDate.textContent = `${day}/${month}/${year}, ${hour}:${min}`;
```

2. Displaying Movement Dates

- **Objective**: Display the date for each movement (transaction) in the application.
- **Steps**:
  - Modify the `displayMovements` function to include dates for each transaction.
  - Loop over the `movements` and `movementDates` arrays simultaneously using the `forEach` loop and the index.
  - Extract and format the date for each movement and display it alongside the movement in the UI.

```
movements.forEach((mov, i) => {
  const date = new Date(acc.movementDates[i]);
  const day = `${date.getDate()}`.padStart(2, "0");
  const month = `${date.getMonth() + 1}`.padStart(2, "0");
  const year = date.getFullYear();

  const displayDate = `${day}/${month}/${year}`;

  const html = `
    <div class="movements__row">
      <div class="movements__type movements__type--${type}">${
    i + 1
  } ${type}</div>
      <div class="movements__date">${displayDate}</div>
      <div class="movements__value">${mov.toFixed(2)}€</div>
    </div>`;
  containerMovements.insertAdjacentHTML("afterbegin", html);
});
```

3. Handling Date Formatting Issues

- **Objective**: Ensure proper formatting for single-digit days and months by padding them with a leading zero.

- **Steps**:
  - Use `padStart(2, '0')` on the day and month strings to ensure two-digit formatting.

4. Updating Dates During Transactions

- **Objective**: Automatically add a date for new transactions like loans and transfers.
- **Steps**:
  - When a new movement is added (e.g., a loan or transfer), push the current date into the `movementDates` array.
  - Format the date correctly using `toISOString()` to ensure consistency.

```
currentAccount.movements.push(amount);
currentAccount.movementDates.push(new Date().toISOString());
```

5. Handling Time Formatting

- **Objective**: Ensure that the hour and minute are correctly formatted with leading zeros if necessary.
- **Steps**:
  - Use the `padStart` method on the hour and minute when displaying the current date and time.

```
const hour = `${now.getHours()}`.padStart(2, "0");
const min = `${now.getMinutes()}`.padStart(2, "0");
```

6. Resolving Errors and Debugging

- **Objective**: Debug any issues, such as undefined properties or incorrect formatting.
- **Steps**:
  - Ensure that when displaying sorted movements, both the movements and movementDates arrays are correctly accessed.
  - Handle potential errors when formatting dates by ensuring that methods like `toISOString()` are correctly called.

Key Takeaways

- **Date Formatting**: Ensure consistency in date formats across the application, particularly when handling different locales.
- **Debugging**: Always test and debug features like date and time formatting, especially when they involve dynamic content such as user transactions.
- **Refactoring**: Implementing changes becomes easier when functions and methods are well-refactored and organized. In this explanation, we covered how to perform calculations with dates in JavaScript, particularly focusing on how to calculate the number of days between two dates. Here's a breakdown of the key points:

1. **Converting Dates to Timestamps:**

- When you convert a date to a number in JavaScript, it gives you the timestamp in milliseconds since January 1, 1970.
- This allows for calculations like subtracting one date from another to determine the difference in milliseconds.

2. **Calculating Days Between Two Dates:**

- To find out how many days have passed between two dates, you can subtract one date from another.
- The result will be in milliseconds, so you'll need to convert it to days by dividing it by the number of milliseconds in a day:
  - 1 second = 1000 milliseconds
  - 1 minute = 60 seconds
  - 1 hour = 60 minutes
  - 1 day = 24 hours
  - Therefore, `1 day = 1000 * 60 * 60 * 24 milliseconds`.

```
const calcDaysPassed = (date1, date2) =>
  Math.abs(date2 - date1) / (1000 * 60 * 60 * 24);
```

3. **Handling Negative Values:**

- If you want to ensure that the result is always a positive number, regardless of which date is earlier, you can use the `Math.abs()` function to get the absolute value.

4. **Rounding the Result:**

- In cases where the difference between two dates includes partial days, you might want to round the result to the nearest whole number using `Math.round()`.

5. **Formatting Dates Based on Time Passed:**

- A common use case is to display dates relative to the current date, such as "today," "yesterday," or "X days ago."
- You can implement a function to format dates accordingly:

```
const formatMovementDate = function (date) {
  const daysPassed = Math.round((new Date() - date) / (1000 * 60 * 60 * 24));

  if (daysPassed === 0) return "Today";
  if (daysPassed === 1) return "Yesterday";
  if (daysPassed <= 7) return `${daysPassed} days ago`;

  // Fallback to the full date if more than 7 days have passed
  return new Intl.DateTimeFormat("en-US").format(date);
};
```

6. **Integrating the Function into Your Application:**

- By extracting this functionality into a separate function, you can easily reuse it across different parts of your application.
- In this example, the function is used to format and display transaction dates in a more user-friendly way.

7. **Edge Cases:**

- Consider time-related edge cases, such as daylight saving time changes, which may affect date calculations. For more complex scenarios, using a library like `moment.js` (or its modern alternative `date-fns`) is recommended.

8. **Refactoring and Modularity:**

- The logic for formatting dates was refactored into a separate function (`formatMovementDate`) to keep the code modular and easier to maintain.

9. **Real-Life Application:**

- This kind of date formatting is common in real-world applications, such as social media platforms where posts are shown as "today," "yesterday," or "X days ago."

By understanding and implementing these techniques, you can enhance the user experience in your web applications by displaying dates in a more meaningful and human-readable format.

JavaScript Internationalization API

The JavaScript Internationalization (Intl) API is a powerful tool that allows developers to format numbers, dates, strings, and other locale-specific data types according to the user's preferred language and region. This is especially useful for applications that need to support users from different parts of the world, as formats for currencies, dates, and times can vary widely between countries.

Formatting Dates with the Intl API

In this section, we'll focus on formatting dates using the Intl API. Dates can be displayed in various formats depending on the locale (e.g., U.S., Europe, Asia). This can include different orders of day, month, and year, as well as time formats.

Example Scenario: Formatting Login Date

Imagine you have an application that displays the login date in two places:

1. The main dashboard.
2. Each transaction or movement.

Here's how you can use the Intl API to format these dates according to different locales.

Step-by-Step Implementation

1. **Initialize Date Formatting:**

- You can create a date formatter using `Intl.DateTimeFormat()`, passing a locale string that specifies the language and country (e.g., `"en-US"` for U.S. English).
- The `format()` method on this formatter will return the date in the specified format.

```
const now = new Date();
const formattedDate = new Intl.DateTimeFormat("en-US").format(now);
console.log(formattedDate); // Example: 8/12/2020
```

2. **Customize Date Formatting:**

- You can add options to the `DateTimeFormat` function to include time, specify numeric vs. long month names, or include the weekday.

```
const options = {
  hour: "numeric",
  minute: "numeric",
  day: "numeric",
  month: "long",
  year: "numeric",
  weekday: "long",
};
const formattedDateWithTime = new Intl.DateTimeFormat(
  "en-US",
  options
).format(now);
console.log(formattedDateWithTime); // Example: Wednesday, August 12, 2020,
1:23 PM
```

3. **Get Locale from User's Browser:**

- Instead of hardcoding the locale, you can retrieve it from the user's browser settings using `navigator.language`.

```
const userLocale = navigator.language;
const formattedDate = new Intl.DateTimeFormat(userLocale, options).format(
  now
);
```

4. **Apply Locale to User-Specific Data:**

- If your application stores the user's preferred locale, you can apply it when formatting dates.

```
const user = { locale: "pt-PT" }; // Example: Portuguese (Portugal)
const formattedDate = new Intl.DateTimeFormat(user.locale, options).format(
  now
);
```

Practical Example: Displaying Dates in Different Locales

- **User 1 (Portuguese):** Dates are formatted according to Portuguese conventions (e.g., "12 de agosto de 2020").
- **User 2 (American):** Dates are formatted according to U.S. conventions (e.g., "August 12, 2020").

```
const currentAccount = { locale: "pt-PT" }; // User's locale
const formattedDate = new Intl.DateTimeFormat(
  currentAccount.locale,
  options
).format(now);
console.log(formattedDate); // Output: "quarta-feira, 12 de agosto de 2020, 13:23"
```

This demonstrates how easy it is to make your application adaptable to users from around the world by using the Intl API.

Key Takeaways

- The Intl API is essential for formatting locale-specific data like dates and numbers.
- You can customize formats using options like `weekday`, `year`, `month`, `day`, `hour`, and `minute`.
- Locale can be dynamically set based on user preferences or browser settings.
- This approach ensures that your application is accessible and user-friendly, regardless of where your users are located. Here's a detailed breakdown of the lecture content, focusing on formatting numbers using the Internationalization (Intl) API:

Formatting Numbers with the Intl API

1. Introduction to Number Formatting

- **Experimenting in Console**:
  - The Intl API allows for easy number formatting.
  - Example of creating a number and formatting it with different locales (e.g., US, Germany, Syria).
  - **US Example**:

```
new Intl.NumberFormat("en-US").format(3884000);
```

    - Results in `3,884,000`, using commas as thousand separators.

2. Locale-Specific Formatting

- **Germany Example**:
  - In Germany, the formatting is different:

```
new Intl.NumberFormat("de-DE").format(3884000);
```

    - Results in `3.884.000`, with periods as thousand separators and a comma for decimals.
- **Syria Example**:
    - Arabic formatting:

```
new Intl.NumberFormat("ar-SY").format(3884000);
```

        - The number format is specific to Arabic numerals.

3. Using Browser's Locale

- **Browser Locale Example**:

```
new Intl.NumberFormat(navigator.language).format(3884000);
```

    - Uses the browser's default locale to format the number.
    - Example: In the UK, formatting resembles the US format.

4. Advanced Formatting with Options

- **Formatting with Units**:
    - **Example**:

```
new Intl.NumberFormat("en-US", {
  style: "unit",
  unit: "mile-per-hour",
}).format(50);
```

        - Formats the number as `50 mph`, with unit `mile-per-hour`.
- **Percentage Formatting**:
    - **Example**:

```
new Intl.NumberFormat("en-US", { style: "percent" }).format(0.5);
```

        - Formats as `50%`.
- **Currency Formatting**:
    - **Example**:

```
new Intl.NumberFormat("en-US", {
  style: "currency",
  currency: "USD",
}).format(1000);
```

    - Results in `$1,000.00`.
  - The position of the currency symbol varies by locale (e.g., € in Europe comes after the number).

5. Implementing in a Web Application

- **Formatting Movements**:
  - Example of replacing manual formatting in the application with Intl API formatting.
  - **Old Method**: Manually adding currency symbols and formatting.
  - **New Method**:

```
const formattedMovement = new Intl.NumberFormat(acc.locale, {
  style: "currency",
  currency: acc.currency,
}).format(mov);
```

  - This method adapts to the user's locale and currency settings.

6. Creating a Reusable Function

- **Refactoring for Reusability**:
  - Create a reusable function to format any value with given locale and currency.
  - **Example**:

```
function formatCur(value, locale, currency) {
  return new Intl.NumberFormat(locale, {
    style: "currency",
    currency: currency,
  }).format(value);
}
```

  - This function can be reused across the application for consistent formatting.

7. Applying to Other Parts of the Application

- **Formatting Balances and Statistics**:
  - Applied the `formatCur` function to format balances and other financial figures in the application.
  - Ensured consistency across all parts of the application.

8. Key Takeaways

- **Intl API** is a powerful tool for formatting numbers, dates, and more, based on locale.
- **Consistency**: By using Intl API, developers can ensure that number formatting is consistent across different parts of an application.
- **Flexibility**: The API provides options to format numbers as currencies, percentages, units, etc., making it versatile for different applications.
- **Browser Locale**: The API can automatically adapt to the user's browser settings for locale-specific formatting.

Here's a detailed breakdown of the lecture content, focusing on formatting numbers using the Internationalization (Intl) API:

Formatting Numbers with the Intl API

1. Introduction to Number Formatting

- **Experimenting in Console**:
  - The Intl API allows for easy number formatting.
  - Example of creating a number and formatting it with different locales (e.g., US, Germany, Syria).
  - **US Example**:

```
new Intl.NumberFormat("en-US").format(3884000);
```

    - Results in `3,884,000`, using commas as thousand separators.

2. Locale-Specific Formatting

- **Germany Example**:
  - In Germany, the formatting is different:

```
new Intl.NumberFormat("de-DE").format(3884000);
```

    - Results in `3.884.000`, with periods as thousand separators and a comma for decimals.
- **Syria Example**:
  - Arabic formatting:

```
new Intl.NumberFormat("ar-SY").format(3884000);
```

    - The number format is specific to Arabic numerals.

3. Using Browser's Locale

- **Browser Locale Example**:

```
new Intl.NumberFormat(navigator.language).format(3884000);
```

- Uses the browser's default locale to format the number.
- Example: In the UK, formatting resembles the US format.

4. Advanced Formatting with Options

- **Formatting with Units**:
  - **Example**:

```
new Intl.NumberFormat("en-US", {
  style: "unit",
  unit: "mile-per-hour",
}).format(50);
```

    - Formats the number as `50 mph`, with unit `mile-per-hour`.

- **Percentage Formatting**:
  - **Example**:

```
new Intl.NumberFormat("en-US", { style: "percent" }).format(0.5);
```

    - Formats as `50%`.

- **Currency Formatting**:
  - **Example**:

```
new Intl.NumberFormat("en-US", {
  style: "currency",
  currency: "USD",
}).format(1000);
```

    - Results in `$1,000.00`.
  - The position of the currency symbol varies by locale (e.g., € in Europe comes after the number).

5. Implementing in a Web Application

- **Formatting Movements**:
  - Example of replacing manual formatting in the application with Intl API formatting.
  - **Old Method**: Manually adding currency symbols and formatting.
  - **New Method**:

```
const formattedMovement = new Intl.NumberFormat(acc.locale, {
  style: "currency",
  currency: acc.currency,
}).format(mov);
```

- This method adapts to the user's locale and currency settings.

6. Creating a Reusable Function

- **Refactoring for Reusability**:
  - Create a reusable function to format any value with given locale and currency.
  - **Example**:

```
function formatCur(value, locale, currency) {
  return new Intl.NumberFormat(locale, {
    style: "currency",
    currency: currency,
  }).format(value);
}
```

  - This function can be reused across the application for consistent formatting.

7. Applying to Other Parts of the Application

- **Formatting Balances and Statistics**:
  - Applied the `formatCur` function to format balances and other financial figures in the application.
  - Ensured consistency across all parts of the application.

8. Key Takeaways

- **Intl API** is a powerful tool for formatting numbers, dates, and more, based on locale.
- **Consistency**: By using Intl API, developers can ensure that number formatting is consistent across different parts of an application.
- **Flexibility**: The API provides options to format numbers as currencies, percentages, units, etc., making it versatile for different applications.
- **Browser Locale**: The API can automatically adapt to the user's browser settings for locale-specific formatting.

Timers in JavaScript

JavaScript provides two types of timers that are essential for handling time-based events:

1. `setTimeout`: Executes a function once after a specified delay.
2. `setInterval`: Executes a function repeatedly at regular intervals until stopped.

`setTimeout`

The `setTimeout` function allows us to execute code at a later time, simulating delays like waiting for a pizza delivery. Here's how it works:

```
setTimeout(() => {
  console.log("Here is your pizza 🍕");
}, 3000);
```

- **Callback Function**: The first argument to `setTimeout` is a callback function, which is called after the specified delay.
- **Delay**: The second argument is the delay in milliseconds (e.g., `3000` for 3 seconds).

Important Note: The code execution does not stop at `setTimeout`. JavaScript registers the callback function to be called later and continues executing the subsequent code immediately.

```
console.log("Waiting...");
setTimeout(() => {
  console.log("Here is your pizza 🍕");
}, 3000);
```

Output:

```
Waiting...
(After 3 seconds)
Here is your pizza 🍕
```

This demonstrates the asynchronous nature of JavaScript, where the timer counts down in the background while the rest of the code continues to execute.

Passing Arguments to the Callback

Passing arguments to the callback function in `setTimeout` can be tricky because we don't call the function ourselves. However, `setTimeout` allows passing additional arguments, which are then passed to the callback function.

```
setTimeout(
  (ingredient1, ingredient2) => {
    console.log(`Here is your pizza with ${ingredient1} and ${ingredient2} 🍕`);
  },
  3000,
  "olives",
  "spinach"
);
```

Output (after 3 seconds):

```
Here is your pizza with olives and spinach 🍕
```

Cancelling a Timeout

You can cancel a `setTimeout` before the delay has passed using `clearTimeout`. This is useful if certain conditions are met (e.g., if the user cancels the pizza order).

```
const ingredients = ["olives", "spinach"];
const pizzaTimer = setTimeout(
  (ingredient1, ingredient2) => {
    console.log(`Here is your pizza with ${ingredient1} and ${ingredient2} 🍕`);
  },
  3000,
  ...ingredients
);

if (ingredients.includes("spinach")) {
  clearTimeout(pizzaTimer);
  console.log("Pizza order canceled due to spinach");
}
```

If the `ingredients` array contains `"spinach"`, the timer is cleared, and the pizza message won't be logged.

Key Takeaways:

- `setTimeout` is for one-time delayed execution.
- The callback function is passed as an argument and is executed after the specified delay.
- **Asynchronous JavaScript**: Code continues executing immediately after `setTimeout` is called.
- `clearTimeout` can cancel the scheduled function before it executes.

This concludes the basic understanding of timers in JavaScript!

`setInterval` in JavaScript

While `setTimeout` is used to run a function once after a specified delay, `setInterval` is used to execute a function repeatedly at regular intervals.

Using `setInterval` to Create a Clock

Here's how you can use `setInterval` to create a simple clock that logs the current date and time to the console every second:

```
setInterval(() => {
  const now = new Date();
```

```
  console.log(now);
}, 1000); // Executes every 1 second (1000 milliseconds)
```

- **Callback Function**: The callback function is executed repeatedly at the interval specified in milliseconds.
- **Interval**: The second argument defines the interval duration (e.g., `1000` for 1 second).

When this code runs, the current time is logged to the console every second, continuously updating to reflect the current time.

Correcting the Earlier Mistake

Initially, we mistakenly used `setTimeout` instead of `setInterval`. While `setTimeout` executes a function once after a delay, `setInterval` repeatedly executes the function at the specified interval. By switching to `setInterval`, we get the desired behavior of a function running every second.

```
setInterval(() => {
  const now = new Date();
  console.log(now);
}, 1000);
```

This will log the current date and time every second, creating a simple clock in the console.

Customizing the Interval

You can adjust the interval duration to suit different needs. For example, changing the interval to `3000` milliseconds will log the time every 3 seconds:

```
setInterval(() => {
  const now = new Date();
  console.log(now);
}, 3000); // Executes every 3 seconds
```

Challenge: Building a Real Clock

A nice exercise is to create a clock that logs only the current hours, minutes, and seconds. Here's a hint on how you can achieve this:

1. Use the `getHours()`, `getMinutes()`, and `getSeconds()` methods of the `Date` object to extract the respective time components.
2. Format the output to show a nicely formatted time (e.g., `14:30:45`).

Here's a starting point:

```

```

```
setInterval(() => {
  const now = new Date();
  const hours = now.getHours();
  const minutes = now.getMinutes();
  const seconds = now.getSeconds();
  console.log(`${hours}:${minutes}:${seconds}`);
}, 1000); // Executes every 1 second
```

Key Takeaways:

- `setInterval` is used for repeated execution at a defined interval.
- The interval duration is specified in milliseconds.
- The `setInterval` function continues to execute until it is explicitly stopped (e.g., using `clearInterval`).

This concludes the discussion on `setInterval` in JavaScript. Feel free to take up the challenge and build your own clock! To implement the logout timer that resets with user activity, we'll extend the functionality you've already started by adding some key steps. Here's a detailed explanation of how you can implement this feature:

1. **Start the Logout Timer on User Login**

- You already created a `startLogoutTimer` function that begins the countdown when the user logs in.
- This timer is displayed as a countdown in minutes and seconds in the UI.

2. **Reset the Timer on User Activity**

- To ensure the timer resets with user activity (such as money transfers or loan requests), you'll need to include the `startLogoutTimer` function in these activities.
- The timer should restart each time an activity is performed.

3. **Global Timer Management**

- The `timer` variable should be globally accessible so that it can be cleared and reset across different functions and user activities.
- This ensures that only one timer runs at a time, and the timer is correctly reset upon new activity.

4. **Clear the Previous Timer on New Activity**

- Before starting a new timer, check if an existing timer is running. If so, clear it to avoid multiple timers running simultaneously.

5. **Apply the Timer Reset in User Actions**

- Apply the timer reset logic in key user actions such as transfers and loan requests.

Here's how you can implement it in code:

Implementation Steps

1. **Global Timer Variable:**

```
let timer;
```

2. **Function to Start and Reset the Timer:**

```
function startLogoutTimer() {
  let time = 120; // Set the timer for 2 minutes or as required

  const tick = function () {
    const minutes = String(Math.trunc(time / 60)).padStart(2, "0");
    const seconds = String(time % 60).padStart(2, "0");

    // Display the remaining time in the UI
    document.querySelector(".timer").textContent = `${minutes}:${seconds}`;

    // Log out the user when time reaches 0
    if (time === 0) {
      clearInterval(timer);
      document.querySelector(".app").style.opacity = 0;
      document.querySelector(".welcome").textContent =
        "Log in to get started";
    }

    // Decrease time by 1 second
    time--;
  };

  // Start the timer immediately and call it every second
  tick();
  timer = setInterval(tick, 1000);
  return timer;
}
```

3. **Reset the Timer on User Activity:**

   - For example, reset the timer after a transfer:

```
function transferMoney() {
  // Clear the existing timer
  if (timer) clearInterval(timer);

  // Logic for transferring money
  // ...

  // Restart the logout timer
  timer = startLogoutTimer();
}
```

- Similarly, reset the timer after a loan request:

```
function requestLoan() {
  // Clear the existing timer
  if (timer) clearInterval(timer);

  // Logic for requesting a loan
  // ...

  // Restart the logout timer
  timer = startLogoutTimer();
}
```

4. **Logging Out After Timer Expires:**
    - Ensure that when the timer reaches zero, the user is logged out and the UI is reset to its initial state.

Final Notes

- **Usability:** The timer now resets with every user action, reflecting real-world bank application behavior.
- **Performance:** Clearing and resetting the timer on each activity ensures efficient timer management, preventing multiple overlapping timers.
- **Security:** This approach enhances the security by automatically logging users out after a period of inactivity, aligning with best practices for sensitive applications.

Section 8: Behind the scenes JavaScript

Introduction to JavaScript: A High-Level Overview

In this section, we'll dive into a high-level overview of JavaScript, giving you a glimpse into the vast landscape of topics we'll cover as we progress through the course. The goal here is to provide a broad understanding of the key concepts, setting the stage for more detailed explorations later. This is going to be a long and intense ride, so buckle up!

What is JavaScript?

JavaScript is often described as a high-level, object-oriented, multi-paradigm programming language. But that description is just the tip of the iceberg. Here's a more comprehensive, though admittedly complex, definition: JavaScript is a high-level, prototype-based, object-oriented, multi-paradigm, interpreted or just-in-time compiled, dynamic, single-threaded, garbage-collected programming language with first-class functions and a non-blocking event loop concurrency model.

While this might sound overwhelming, it's a great way to introduce the core concepts we'll be unpacking throughout the course.

Unpacking the JavaScript Definition

To make sense of this definition, let's break it down into nine key topics:

1. **High-Level Language:**
    - **Abstractions:** In high-level languages like JavaScript and Python, developers don't need to manage hardware resources like memory or CPU manually. These languages provide abstractions that handle these tasks, making them easier to learn and use. However, this convenience comes with a trade-off: programs written in high-level languages may not be as fast or optimized as those written in low-level languages like C.
    - **Garbage Collection:** JavaScript includes a built-in garbage collector, an algorithm that automatically removes old, unused objects from memory, freeing up resources without requiring manual intervention.
2. **Interpreted or Just-In-Time Compiled Language:**
    - **Machine Code Translation:** All programs ultimately need to be translated into machine code (zeros and ones) that the computer's processor can understand. In JavaScript, this translation happens within the JavaScript engine, either through interpretation or just-in-time (JIT) compilation.
3. **Multi-Paradigm Language:**
    - **Programming Paradigms:** JavaScript supports multiple programming paradigms, including procedural, object-oriented, and functional programming. A paradigm is a way of structuring code, and JavaScript's flexibility allows developers to choose the most suitable paradigm for their projects.
    - **Imperative vs. Declarative Paradigms:** JavaScript can also be categorized into imperative and declarative paradigms, but we'll explore this in more detail later.
4. **Prototype-Based Object-Oriented Language:**
    - **Prototypal Inheritance:** In JavaScript, objects inherit properties and methods from prototypes. For example, when you create an array and use methods like `push()`, you're utilizing the concept of prototypal inheritance, where the array inherits these methods from a prototype or blueprint.
5. **First-Class Functions:**
    - **Functions as Variables:** In JavaScript, functions are treated as first-class citizens, meaning they can be stored in variables, passed as arguments to other functions, and returned from functions. This capability is a cornerstone of functional programming in JavaScript.
6. **Dynamic Language:**
    - **Dynamically-Typed:** JavaScript is dynamically-typed, meaning you don't need to declare the data type of a variable. The type is determined at runtime, and variables can change types as they are reassigned. While this offers flexibility, it can also lead to bugs, which is why some developers prefer using TypeScript, a strongly-typed superset of JavaScript.
7. **Single-Threaded:**

- **Single-Threaded Execution:** JavaScript runs on a single thread, meaning it can only execute one task at a time. However, it uses a non-blocking event loop to handle multiple tasks concurrently without blocking the main thread.

8. **Non-Blocking Event Loop Concurrency Model:**

    - **Handling Multiple Tasks:** The event loop allows JavaScript to perform long-running tasks, like fetching data from a server, in the background without blocking the main thread. Once the task is complete, the event loop returns the result to the main thread, enabling non-blocking behavior.

JavaScript Engine

A **JavaScript engine** is a computer program responsible for executing JavaScript code. The engine interprets and compiles the JavaScript source code into machine code that the processor can execute. Every web browser has its own JavaScript engine, with Google's V8 engine being one of the most famous. V8 powers both Google Chrome and Node.js, allowing JavaScript to run on the server side, outside the browser.

Components of a JavaScript Engine

- **Call Stack**: Where the JavaScript code is executed. It handles execution contexts, managing function calls, and keeping track of where the code is in its execution process.
- **Heap**: An unstructured memory pool that stores objects and data needed by the application.

Compilation and Interpretation

Before JavaScript code can be executed, it must be converted into machine code, which is understood by the computer's processor. This can be done through:

- **Compilation**: The entire source code is converted into machine code in one go and stored in a portable file. This file can be executed later, allowing for faster performance as the code has already been compiled.
- **Interpretation**: The source code is read and executed line by line, with the conversion to machine code happening just before execution. This method is slower but was traditionally used in JavaScript.

Just-In-Time Compilation

Modern JavaScript engines use a combination of both compilation and interpretation called **Just-In-Time (JIT) Compilation**. In JIT, the code is compiled into machine code right before execution. This approach allows for faster performance, similar to compiled languages, while still offering the flexibility of interpretation. The engine starts by creating an unoptimized version of the machine code for quick execution and then continues optimizing and recompiling the code during runtime, leading to better performance.

Abstract Syntax Tree (AST)

When JavaScript code enters the engine, it is first parsed into an **Abstract Syntax Tree (AST)**, a data structure representing the structure of the code. The AST is generated by breaking the code into meaningful components and organizing them into a tree structure. This tree is then used for compiling the code into machine code.

JavaScript Runtime

A **JavaScript runtime** is an environment in which JavaScript code is executed. It includes everything needed to run JavaScript, including the JavaScript engine, Web APIs, and other necessary tools.

Components of a JavaScript Runtime

- **JavaScript Engine**: The core component responsible for executing the code.
- **Web APIs**: Functions and features provided by the browser, such as DOM manipulation, timers, and console logging. These are not part of the JavaScript language itself but are accessible through the global `window` object in browsers.
- **Callback Queue**: A data structure holding callback functions that are ready to be executed. These callbacks are queued after events (like a button click) occur and are then executed by the engine when the call stack is empty.
- **Event Loop**: A mechanism that continuously checks the callback queue and moves callbacks into the call stack for execution when the stack is empty. This allows JavaScript to handle asynchronous events efficiently, making it non-blocking.

Browser vs. Node.js Runtime

- **Browser JavaScript Runtime**: Includes the engine, Web APIs provided by the browser, a callback queue, and the event loop.
- **Node.js JavaScript Runtime**: Includes the engine but replaces Web APIs with C++ bindings and a thread pool since it's not running in a browser environment.

How JavaScript Code is Executed

Introduction to Execution Contexts and the Call Stack:

When JavaScript code is ready to be executed after compilation, it is executed within an environment called an *execution context*. The first type of context that is created is the *global execution context*, which is responsible for executing *top-level code*. Top-level code refers to code that is not inside any function.

Global Execution Context:

- The global execution context is the default context where all top-level code executes. In this context, variables, functions, and other top-level code elements are declared and initialized.
- Functions are also declared in this context, but the code inside these functions does not execute until the functions are explicitly called.
- For instance, consider the following code snippet:

```
const name = "John Doe";

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

const x = greet();
```

- The variable `name` is top-level code and is declared and initialized in the global execution context.
- The function `greet` is declared but not executed until it is called later in the code.

What is an Execution Context?

- An execution context is an abstract environment where JavaScript code is executed. It includes all necessary information required for code execution, such as local variables, function arguments, and other data.
- The execution context is akin to a "box" that holds all the necessary items (e.g., local variables, function arguments) needed to execute a specific piece of code.

Components of an Execution Context:

1. **Variable Environment:**
   - This includes all the variables and function declarations within the current context. It also contains a special `arguments` object, which holds all the arguments passed into a function.
2. **Scope Chain:**
   - The scope chain contains references to variables located outside the current function but accessible within it. This allows functions to access variables from outer scopes.
3. **This Keyword:**
   - The `this` keyword is a special variable that refers to the context in which the code is executed. It is unique for each execution context.

Arrow Functions and Execution Contexts:

- Arrow functions are a special type of function that does not have its own `arguments` object or `this` keyword. Instead, they inherit these from their closest enclosing regular function.

Call Stack:

- The *call stack* is a mechanism that tracks the execution of functions and their respective contexts. Each time a function is called, a new execution context is created and placed (or "pushed") on top of the call stack.
- The function on the top of the stack is the one currently being executed. Once a function completes execution, its context is removed (or "popped") from the stack, and the engine resumes execution of the previous context.
- This process continues until the global execution context is the only one remaining, which persists until the program finishes or the browser tab is closed.

Example Walkthrough:

Consider the following example:

```
const name = "John Doe";
```

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

const x = greet();
```

- **Global Execution Context:**
    - Variables and functions are declared.
    - The `name` variable is initialized.
    - The `greet` function is declared.
- **Function Call:**
    - The function `greet` is called, which creates a new execution context that is pushed onto the call stack.
    - Inside the `greet` function, the `greeting` variable is declared and initialized.
    - The function returns a value, and the execution context is popped off the stack.
- **Return to Global Execution Context:**
    - The returned value is assigned to the variable `x`.

Conclusion:

The JavaScript engine uses the call stack and execution contexts to manage the order and scope of code execution. Each function call creates a new execution context, which is managed through the call stack. Understanding how these work is crucial for grasping how JavaScript executes code and handles functions.

Deep Dive into JavaScript's Scope and Scope Chain

In this lecture, we're going to delve deep into the concept of scope in JavaScript, a crucial topic that governs how variables are organized, accessed, and managed by the JavaScript engine. Understanding scope is key to writing effective and bug-free code. This lecture builds on what we learned previously about execution contexts, and we'll explore how scope and the scope chain work together behind the scenes in JavaScript.

Scoping and Lexical Scoping

- **Scoping** controls the accessibility of variables and functions in your code. It determines where variables "live" and where they can be accessed.
- **Lexical Scoping** means that the scope of a variable is determined by its position in the source code. JavaScript uses lexical scoping, which means that functions are executed using the scope in which they were defined, not the scope from which they are invoked.

Understanding Scope

- **Scope** is the environment in which a variable is declared. In JavaScript, there are three types of scopes:
    1. **Global Scope**: Variables declared outside of any function or block are in the global scope and can be accessed from anywhere in the code.

2. **Function Scope**: Each function creates its own scope, and variables declared within a function are accessible only within that function.
3. **Block Scope**: Introduced in ES6, block scope refers to variables declared within a block (e.g., within `{}`) and are only accessible within that block.

Scope of a Variable

- The **scope of a variable** is the region of code where the variable is accessible. For example, a variable declared inside a function is only accessible within that function (function scope), while a variable declared with `let` or `const` inside a block is only accessible within that block (block scope).

Three Types of Scope in JavaScript

1. **Global Scope**:
   - Variables declared in the global scope can be accessed from anywhere in the program.
   - Example:

```
let globalVar = "I'm global";

function printGlobalVar() {
  console.log(globalVar);
}
printGlobalVar(); // Outputs: I'm global
```

2. **Function Scope**:
   - Variables declared within a function are accessible only within that function.
   - Example:

```
function myFunction() {
  let localVar = "I'm local";
  console.log(localVar);
}
myFunction(); // Outputs: I'm local
console.log(localVar); // ReferenceError: localVar is not defined
```

3. **Block Scope**:
   - Variables declared with `let` or `const` within a block `{}` are only accessible within that block.
   - Example:

```
if (true) {
  let blockVar = "I'm block scoped";
  console.log(blockVar); // Outputs: I'm block scoped
```

```
}
console.log(blockVar); // ReferenceError: blockVar is not defined
```

Scope Chain

- The **Scope Chain** is the mechanism by which JavaScript resolves variables. When a variable is accessed, the JavaScript engine looks for the variable in the current scope. If it can't find it, it looks up the scope chain to the parent scopes, continuing until it reaches the global scope.
- **Variable Lookup**: When a variable is not found in the current scope, JavaScript will look up the scope chain to find the variable. If it still can't find it, a reference error will occur.

Example of Scope Chain

- Consider this nested function example:

```
let myName = "Jonas";

function first() {
  let age = 30;

  function second() {
    let job = "teacher";
    console.log(`${myName} is a ${age}-year-old ${job}`);
  }

  second();
}

first();
```

  - **Global Scope**: Contains `myName`.
  - **First Function Scope**: Contains `age`.
  - **Second Function Scope**: Contains `job`.
  - The `second` function can access `myName` and `age` because they are in its parent scopes (global and first function scope). This is the scope chain in action.

Block Scope in ES6

- In ES6, `let`, `const`, and functions are block-scoped, meaning they are confined to the block they are declared in.
- Variables declared with `var` are not block-scoped but function-scoped, meaning they can be accessed outside the block if declared within a function.
- **Example**:

```
if (true) {
  let blockScopedVar = "I'm block scoped";
  var functionScopedVar = "I'm function scoped";
```

```
}
console.log(blockScopedVar); // ReferenceError
console.log(functionScopedVar); // Outputs: I'm function scoped
```

Scope Chain vs. Call Stack

- **Call Stack**: The order in which functions are called during the execution of a program. Each function call creates a new execution context on the call stack.
- **Scope Chain**: The order in which scopes are searched for variables. It is determined by the placement of functions in the code (lexical scoping), not the order in which they are called.

Key Takeaways

1. **Scoping** controls where variables can be accessed.
2. **Lexical Scoping** means the scope of a variable is determined by its position in the code.
3. **Global, Function, and Block Scopes** define where variables can live and be accessed.
4. **Scope Chain** allows inner functions to access variables from outer functions or the global scope.
5. **Block Scoping** with `let` and `const` was introduced in ES6, making scoping more intuitive for developers familiar with other programming languages.

Understanding these concepts is essential for mastering JavaScript and avoiding common pitfalls related to variable access and management. In the next coding lecture, we'll apply these concepts to real code, solidifying your understanding through practical examples.

This lecture is an excellent walkthrough of how scoping works in JavaScript, demonstrating key concepts through practical examples. Below is a summary of the content, focusing on the theory, examples, key takeaways, and important concepts discussed.

Understanding Scoping in JavaScript

Theory and Definitions

- **Scope**: The context in which variables and functions are accessible in a program. Scoping determines the accessibility of variables, and it's crucial for managing the visibility and lifecycle of variables in your code.
- **Global Scope**: Variables defined outside any function or block. Accessible from anywhere in the code.
- **Function Scope**: Variables defined within a function. Only accessible within that function.
- **Block Scope**: Variables defined within a block (`{}`). Introduced in ES6 with `let` and `const`. Accessible only within that block.
- **Scope Chain**: When a variable is not found in the current scope, JavaScript looks up the chain to the parent scope until it finds the variable or reaches the global scope.

Examples

1. **Global and Function Scope:**

```
function calcAge(birthYear) {
  const age = 2024 - birthYear;
  console.log(firstName); // Accesses global variable
  return age;
}

const firstName = "Jonas";
calcAge(1991);
```

- **Explanation**: `firstName` is a global variable. Inside `calcAge`, JavaScript looks for `firstName` in the current scope (function scope). Not finding it there, it looks up to the global scope and finds it.

2. **Nested Function and Scope Chain:**

```
function calcAge(birthYear) {
  const age = 2024 - birthYear;

  function printAge() {
    const output = `You are ${age}, born in ${birthYear}`;
    console.log(output);
  }

  printAge();
}

calcAge(1991);
```

- **Explanation**: The `printAge` function accesses variables `age` and `birthYear` from its parent function `calcAge`, demonstrating the scope chain. These variables are not found in `printAge`'s scope but are found in the parent scope.

3. **Block Scope:**

```
if (birthYear >= 1981 && birthYear <= 1996) {
  const str = "Oh, and you're a millennial!";
  console.log(str); // Works fine
}
console.log(str); // ReferenceError: str is not defined
```

- **Explanation**: `str` is block-scoped (because of `const`). It is only accessible within the `if` block. Outside, it's not defined.

4. **Function Scoping with `var`:**

```
if (birthYear >= 1981 && birthYear <= 1996) {
  var millennial = true;
}
console.log(millennial); // Works, because `var` is function-scoped
```

- **Explanation**: `millennial` is declared with `var`, which ignores block scoping and is instead scoped to the nearest function or global scope.

5. **Block-Scoped Functions:**

```
{
  function add(a, b) {
    return a + b;
  }
}
add(2, 3); // ReferenceError: add is not defined (in strict mode)
```

- **Explanation**: In ES6 and strict mode, functions are block-scoped, making them accessible only within the block where they are defined.

6. **Variable Shadowing and Reassignment:**

```
function printAge() {
  let output = `You are ${age}, born in ${birthYear}`;

  if (birthYear >= 1981 && birthYear <= 1996) {
    let output = "Oh, and you're a millennial!";
    console.log(output); // "Oh, and you're a millennial!"
  }

  console.log(output); // "You are 33, born in 1991"
}
```

- **Explanation**: The `output` variable is redefined in the block scope, shadowing the outer `output` variable. Inside the block, the new `output` is used, but outside, the original is intact.

Key Takeaways

- **Global vs. Local**: Understand where your variables are accessible.
- **Scope Chain**: JavaScript searches for variables from the innermost scope outwards.
- **Block Scope**: `let` and `const` are block-scoped, meaning they only exist within the block they are declared.
- **Function Scope**: Functions, especially with `var`, are scoped to the nearest function, not blocks.
- **Shadowing**: Variables in an inner scope can have the same name as those in an outer scope without conflict, but this can lead to confusion.

Important Concepts

- **Strict Mode**: Enabling strict mode changes some behaviors, like making functions block-scoped.
- **Variable Lookup**: Understanding how JavaScript looks for variables up the scope chain is essential for debugging and writing predictable code.

This detailed breakdown provides a solid understanding of scoping in JavaScript, covering theory, practical examples, and key takeaways to solidify your grasp on the topic.

Hoisting in JavaScript: Understanding the Concept

Introduction to Hoisting

- Hoisting is a fundamental yet often misunderstood concept in JavaScript.
- It allows certain types of variables and functions to be used before they are actually declared in the code.
- Commonly misunderstood as "lifting" or "moving" variables to the top of their scope, hoisting actually involves the creation of a reference in the memory during the **creation phase** of the execution context, before the code is executed.

The Execution Context

- Every execution context in JavaScript consists of three parts:
    1. **Variable Environment**
    2. **Scope Chain**
    3. **This Keyword**
- Hoisting specifically pertains to the **Variable Environment**, where variables and functions are stored before code execution begins.

Hoisting of Different Variable Types

1. **Function Declarations**
    - Function declarations are fully hoisted, meaning they are stored in the Variable Environment with their entire function body.
    - **Practical Implication:** Function declarations can be used before they are declared in the code.
2. **Variables Declared with `var`**
    - Variables declared with `var` are also hoisted, but unlike function declarations, they are initialized with `undefined`.
    - **Practical Implication:** If you try to use a `var` variable before its declaration, you will get `undefined`, not an error. This can lead to unexpected behavior and bugs.
3. **Variables Declared with `let` and `const`**
    - Technically, `let` and `const` variables are also hoisted, but they are placed in a **Temporal Dead Zone (TDZ)** until their actual declaration.
    - **Practical Implication:** Accessing `let` or `const` variables before their declaration results in a `ReferenceError`. This behavior prevents the use of uninitialized variables, reducing potential

bugs.

4. **Function Expressions and Arrow Functions**
    - Hoisting for function expressions and arrow functions depends on how they are declared:
        - If declared with `var`, they behave like `var` variables (hoisted to `undefined`).
        - If declared with `let` or `const`, they follow the same rules as `let` and `const` variables, including being in the TDZ.

Temporal Dead Zone (TDZ)

- The TDZ is the time between entering a scope and the actual declaration of a variable within that scope.
- **Example:**

```
{
  console.log(job); // ReferenceError: Cannot access 'job' before
initialization
  const job = "developer";
}
```

- **TDZ Behavior:**
    - Before the `job` variable is declared, it is in the TDZ, making it inaccessible.
    - Once declared, the variable can be safely used.

Why Does Hoisting Exist?

- **For Functions:** Hoisting allows function declarations to be used before they are defined, which is essential for certain programming patterns like mutual recursion.
- **For `var` Variables:** Hoisting `var` variables to `undefined` was an unintended side effect of the initial implementation of function hoisting. While it made sense during JavaScript's early development, it has since been considered problematic.

Dealing with Hoisting Issues

- Modern JavaScript largely avoids `var` in favor of `let` and `const` to circumvent hoisting issues.
- By using `let` and `const`, developers can avoid unexpected `undefined` values and potential bugs related to hoisting.

Conclusion

- Hoisting is a critical concept to grasp for effective JavaScript development.
- Understanding how different types of variables and functions are hoisted, and how the TDZ operates, is crucial for writing robust, bug-free code.
- In modern JavaScript, the best practice is to use `let` and `const` to avoid common pitfalls associated with hoisting.

This transcript covers a comprehensive explanation of hoisting in JavaScript, touching upon the behavior of variables and functions when accessed before their declaration.

Key Concepts:

1. **Hoisting in Variables:**
    - Variables declared with `var` are hoisted to the top and initialized with `undefined`.
    - `let` and `const` variables are also hoisted, but they remain uninitialized in the Temporal Dead Zone (TDZ) until their declaration is executed, leading to a `ReferenceError` if accessed before initialization.
2. **Hoisting in Functions:**
    - **Function Declarations:** These are hoisted completely, allowing them to be called before their actual code definition.
    - **Function Expressions and Arrow Functions:** When defined using `var`, they behave like variables, being hoisted and initialized with `undefined`, leading to errors if invoked before initialization.

Code Examples:

- **Variables:**

```
console.log(me); // undefined
console.log(job); // ReferenceError
console.log(year); // ReferenceError

var me = "Uday";
let job = "Teacher";
const year = 1991;
```

- **Functions:**

```
console.log(addDecl(2, 3)); // 5
console.log(addExpr(2, 3)); // ReferenceError
console.log(addArrow(2, 3)); // ReferenceError

function addDecl(a, b) {
  return a + b;
}

const addExpr = function (a, b) {
  return a + b;
};

const addArrow = (a, b) => a + b;
```

Practical Example:

- **Potential Pitfall:**

```
if (!numProducts) deleteShoppingCart();

var numProducts = 10;

function deleteShoppingCart() {
  console.log("All products deleted");
}
```

Output: "All products deleted"

- Due to hoisting, `numProducts` is initially `undefined`, which is falsy, triggering the deletion.

Best Practices:

- Avoid using `var`. Prefer `const` for variables that don't change and `let` for those that do.
- Declare all variables at the top of their scope.
- Define functions before invoking them, even function declarations, for cleaner code.

Additional Insights:

- Variables declared with `var` create properties on the global `window` object, unlike `let` and `const`.

This transcript effectively covers the crucial aspects of hoisting, emphasizing common pitfalls and best practices to avoid bugs related to variable and function hoisting in JavaScript. The `this` keyword is a fundamental concept in JavaScript, and understanding how it works is crucial for writing effective and bug-free code. Here's a summary of the key points covered:

The `this` Keyword Overview:

- The `this` keyword is a special variable created for every execution context (function).
- It is one of the three components of an execution context, alongside the variable environment and scope chain.
- **Dynamic Nature**: The value of `this` is not static; it depends on how the function is called, not where or how it is defined.

Rules for Determining `this`:

1. **Method Invocation**:
   - When a function is called as a method (i.e., a function attached to an object), `this` points to the object that is calling the method.
   - Example:

```
const jonas = {
  year: 1989,
  calcAge: function () {
    console.log(this.year); // `this` refers to `jonas`
  },
```

```
};
jonas.calcAge(); // 1989
```

2. **Regular Function Invocation**:
   - When a function is called as a standalone function (not attached to an object), `this` is `undefined` in strict mode.
   - In non-strict mode, `this` points to the global object (e.g., `window` in browsers).
   - Example:

```
function showYear() {
  console.log(this.year); // `this` is `undefined` in strict mode
}
showYear();
```

3. **Arrow Functions**:
   - Arrow functions do not have their own `this`. Instead, they inherit `this` from their surrounding (lexical) context.
   - This is known as the "lexical `this`."
   - Example:

```
const jonas = {
  year: 1989,
  calcAge: function () {
    const innerFunction = () => {
      console.log(this.year); // `this` refers to `jonas`, not the
arrow function itself
    };
    innerFunction();
  },
};
jonas.calcAge(); // 1989
```

4. **Event Listeners**:
   - When a function is used as an event listener, `this` refers to the DOM element that the event handler is attached to.
   - Example:

```
const button = document.querySelector("button");
button.addEventListener("click", function () {
  console.log(this); // `this` refers to the button element
});
```

Common Misconceptions:

- `this` does **not** refer to the function itself.
- `this` does **not** point to the variable environment of the function.

Additional Scenarios:

- The `this` keyword also behaves differently when functions are invoked using `new`, `call`, `apply`, or `bind`. These scenarios will be covered in later lessons.

By following these rules and understanding how `this` behaves in different contexts, you'll be better equipped to avoid common pitfalls in JavaScript. Next, let's practice using the `this` keyword in various scenarios to solidify your understanding.

Understanding the `this` Keyword in JavaScript

The `this` keyword in JavaScript is a core concept that can be confusing for beginners. It's important to understand how `this` is defined and how its value changes depending on how a function is called. Let’s break down the key rules and see them in action.

1. Global Scope

- **Global Scope**: When `this` is used outside of any function (in the global scope), it refers to the global object. In a browser, this is the `window` object.

```
console.log(this); // window
```

2. Regular Function Calls

- **Regular Function Call**: When a function is called normally (not as a method of an object), `this` is `undefined` in strict mode.

```
function calcAge(birthYear) {
  console.log(this); // undefined in strict mode
}

calcAge(1991);
```

- In non-strict mode, `this` would point to the global object (`window` in a browser).

3. Arrow Functions

- **Arrow Functions**: Arrow functions don’t have their own `this`. Instead, `this` is lexically inherited from the surrounding scope (the scope in which the arrow function is defined).

```
const calcAgeArrow = (birthYear) => {
  console.log(this); // window (in global scope)
};

calcAgeArrow(1991);
```

- In the example above, because the arrow function is defined in the global scope, `this` points to `window`.

4. Methods in Objects

- **Method Calls**: When `this` is used inside a method, it refers to the object that is calling the method.

```
const jonas = {
  birthYear: 1991,
  calcAge: function () {
    console.log(this); // jonas object
  },
};

jonas.calcAge();
```

- Here, `this` points to the `jonas` object because it's the object that called the `calcAge` method.

5. Method Borrowing

- **Method Borrowing**: You can copy a method from one object to another. When the method is called on the new object, `this` refers to the new object, not the original one.

```
const matilda = {
  birthYear: 2017,
};

matilda.calcAge = jonas.calcAge;
matilda.calcAge(); // matilda object
```

- Even though the `calcAge` method was originally defined in `jonas`, when it's called on `matilda`, `this` points to `matilda`.

6. Function Reference without Call

- **Function as a Value**: A function in JavaScript is a value. If you store a method in a variable and then call it, `this` will be `undefined` because it's not attached to any object.

```
```

```
const f = jonas.calcAge;
f(); // undefined
```

- Here, `this` is `undefined` because the function `f` is called as a regular function, not as a method of an object.

Key Takeaways

- The value of `this` depends on how a function is called.
- **Global Scope**: `this` points to the global object (`window` in a browser).
- **Regular Function**: `this` is `undefined` in strict mode.
- **Arrow Function**: `this` is lexically inherited from the surrounding scope.
- **Method Call**: `this` points to the object that calls the method.
- **Method Borrowing**: The `this` keyword adapts to the object that calls the method, even if it was originally defined in another object.
- **Function Reference**: When a method is copied to a variable and called, `this` becomes `undefined`.

Understanding these rules will help you avoid common pitfalls and write more predictable JavaScript code.

Pitfalls of the `this` Keyword with Regular Functions and Arrow Functions

1. Arrow Function in Object Methods

- **Scenario:** You define a method using an arrow function in an object and expect it to reference the object itself using `this`.
- **Example:**

```
const jonas = {
  firstName: "Jonas",
  greet: () => {
    console.log(`Hey ${this.firstName}`);
  },
};

jonas.greet(); // Hey undefined
```

- **Explanation:**
  - Arrow functions do not have their own `this` context; they inherit `this` from the parent scope.
  - In the example, the parent scope is the global scope (or the scope in which the object was defined).
  - The `this` keyword in the arrow function refers to the `window` object (or `global` in Node.js), which does not have a `firstName` property, leading to `undefined`.
- **Lesson:** Never use arrow functions as methods in objects if you intend to reference the object itself using `this`.

2. Arrow Functions and Global Scope Variables

- **Scenario:** You declare a variable with `var` in the global scope, and an arrow function in an object method references `this` to access the variable.
- **Example:**

```
var firstName = "Matilda";

const jonas = {
  greet: () => {
    console.log(`Hey ${this.firstName}`);
  },
};

jonas.greet(); // Hey Matilda
```

- **Explanation:**
  - The `var` keyword declares a variable that becomes a property of the global object (`window` in browsers).
  - Since the arrow function's `this` refers to the global object, `this.firstName` becomes `window.firstName`, which is `"Matilda"`.
- **Lesson:** Avoid using `var` and be cautious of the scope when using arrow functions, especially in methods.

3. Regular Function Inside a Method

- **Scenario:** A regular function is defined inside an object method, and you expect `this` to refer to the object inside that function.
- **Example:**

```
const jonas = {
  year: 1991,
  calcAge: function () {
    console.log(2023 - this.year);

    function isMillennial() {
      console.log(this.year >= 1981 && this.year <= 1996);
    }
    isMillennial();
  },
};

jonas.calcAge(); // 32, Cannot read property 'year' of undefined
```

- **Explanation:**
  - The `isMillennial` function is a regular function, and when it is called, its `this` is `undefined` (in strict mode) or the global object (in non-strict mode).
  - This happens because regular functions have their own `this`, which defaults to `undefined` in strict mode.
- **Solution 1: Use a reference to `this` with `self` or `that`:**

```
const jonas = {
  year: 1991,
  calcAge: function () {
    const self = this;
    console.log(2023 - self.year);

    function isMillennial() {
      console.log(self.year >= 1981 && self.year <= 1996);
    }
    isMillennial();
  },
};

jonas.calcAge(); // 32, true
```

- **Solution 2: Use an arrow function:**

```
const jonas = {
  year: 1991,
  calcAge: function () {
    console.log(2023 - this.year);

    const isMillennial = () => {
      console.log(this.year >= 1981 && this.year <= 1996);
    };
    isMillennial();
  },
};

jonas.calcAge(); // 32, true
```

- **Lesson:** Use an arrow function or a reference to `this` when you need to access the parent object's `this` inside a nested function.

4. Arguments Object and Arrow Functions

- **Scenario:** You try to use the `arguments` object inside an arrow function.
- **Example:**

```
const add = (...args) => {
  console.log(arguments);
  return args.reduce((acc, curr) => acc + curr, 0);
};

add(2, 5, 8); // Error: arguments is not defined
```

- **Explanation:**
  - The `arguments` object is only available in regular functions, not in arrow functions.
- **Lesson:** Use the rest parameter (`...args`) to access arguments in arrow functions.

Key Takeaways

- **Arrow Functions:** Do not have their own `this` context or `arguments` object. They inherit `this` from the parent scope and should not be used as object methods when `this` is needed.
- **Regular Functions:** Have their own `this` context and `arguments` object, making them suitable for methods and scenarios where these are required.
- **Best Practices:** Avoid pitfalls by understanding the differences between arrow and regular functions and using them appropriately based on the context and requirements.

Understanding Primitive Types vs. Reference Types in JavaScript

In this section, we explored a key difference between primitive types and reference types in JavaScript, focusing on how they are stored and managed in memory. This concept often confuses beginners but is crucial for understanding JavaScript's behavior, especially when dealing with variables and objects.

Primitives:

- **Example Code:**

```
let age = 30;
let oldAge = age;
age = 31;
console.log(age); // 31
console.log(oldAge); // 30
```

- **Explanation:**
  - **Memory Allocation:** When a primitive value (like `age`) is created, JavaScript assigns a unique identifier to the variable and stores the value at a specific memory address in the call stack.
  - **Value Immutability:** The identifier points to a memory address (e.g., `0001`), and the value at that address is immutable.
  - **Variable Assignment:** When `oldAge` is set equal to `age`, both variables point to the same memory address. However, when `age` is updated to `31`, a new memory address is allocated, leaving `oldAge` pointing to the original address (`0001`).

Reference Types:

- **Example Code:**

```
const me = { name: "Jonas", age: 30 };
const friend = me;
friend.age = 27;
console.log(friend.age); // 27
console.log(me.age); // 27
```

- **Explanation:**
  - **Memory Allocation:** Reference types like objects are stored differently. When the `me` object is created, it's stored in the heap (a large memory pool), and a reference to that object is stored in the call stack.
  - **Reference Sharing:** When `friend` is set equal to `me`, both variables point to the same reference in the heap. Therefore, any modification to `friend` also affects `me` because they reference the same object.
  - **Behavior Implication:** This leads to what might seem like unexpected behavior—changing `friend.age` also changes `me.age` since they are not separate objects but merely different references to the same memory location.

Key Takeaways:

- **Primitive vs. Reference Storage:** Primitives are stored directly in the call stack, whereas objects (reference types) are stored in the heap, with references to them stored in the call stack.
- **Value Immutability:** Primitives are immutable in memory, meaning that their memory address content cannot be changed once set. In contrast, reference types can be modified, even if they are declared with `const`, as long as the reference itself doesn't change.
- **Practical Implications:** Understanding these differences is vital for working effectively with JavaScript, especially in situations involving variable assignment, object manipulation, and memory management.

This explanation provides a foundation for deeper exploration of JavaScript's inner workings, setting the stage for more advanced topics like prototypal inheritance, the event loop, and the DOM.

Example: Understanding Primitive vs. Reference Types in JavaScript

Mutating a Primitive Value

Let's start with a simple example of mutating a primitive value, such as a string. We'll set a `lastName` variable to `"Williams"` and then simulate a scenario where this person gets married and changes their last name. We'll also store the old last name for reference.

```
let lastName = "Williams"; // Initial last name
let oldLastName = lastName; // Store the old last name

lastName = "Davis"; // Person gets married and changes last name
```

```
console.log(lastName); // Output: Davis
console.log(oldLastName); // Output: Williams
```

In this case, everything works as expected because each primitive value is stored in its own memory location in the stack. The `oldLastName` holds the value `"Williams"`, while `lastName` is updated to `"Davis"`.

Mutating an Object (Reference Type)

Next, let's consider an object, which is a reference type in JavaScript. When we copy an object, we are actually copying a reference to the memory location of that object, not the object itself.

```
const jessica = {
  firstName: "Jessica",
  lastName: "Williams",
  age: 27,
};

const marriedJessica = jessica; // Copying the reference

marriedJessica.lastName = "Davis"; // Changing last name

console.log(jessica.lastName); // Output: Davis
console.log(marriedJessica.lastName); // Output: Davis
```

Here, both `jessica` and `marriedJessica` point to the same object in memory. Changing the `lastName` on `marriedJessica` also affects `jessica`, since both variables refer to the same object.

Copying an Object Properly

To create a true copy of an object, we can use `Object.assign()`. This function merges two objects and returns a new one, effectively creating a shallow copy.

```
const jessica2 = {
  firstName: "Jessica",
  lastName: "Williams",
  age: 27,
};

const jessicaCopy = Object.assign({}, jessica2);
jessicaCopy.lastName = "Davis";

console.log(jessica2.lastName); // Output: Williams
console.log(jessicaCopy.lastName); // Output: Davis
```

With `Object.assign`, the original `jessica2` object remains unchanged when we modify `jessicaCopy`.

Shallow Copy vs. Deep Clone

However, `Object.assign()` only creates a shallow copy. If the object contains nested objects or arrays, those nested structures will still refer to the same memory location.

```
jessica2.family = ["Alice", "Bob"];
const jessicaCopy2 = Object.assign({}, jessica2);
jessicaCopy2.family.push("Mary");

console.log(jessica2.family); // Output: ["Alice", "Bob", "Mary"]
console.log(jessicaCopy2.family); // Output: ["Alice", "Bob", "Mary"]
```

Here, both `jessica2` and `jessicaCopy2` share the same `family` array, so modifying it in one affects the other.

Conclusion

To create a deep clone, where all nested objects are also copied, you typically need a more sophisticated approach, such as using a library like Lodash. This ensures that changes to the copied object do not affect the original object.

This distinction between shallow and deep copies is crucial when working with complex data structures in JavaScript. Understanding how JavaScript handles primitive and reference types will help you write more predictable and bug-free code.

Section 9: DataStructures modern operators and Strings

Array Destructuring in JavaScript

Introduction: In this section, we are going to explore array destructuring, a modern JavaScript feature introduced in ES6. We'll also use a simulated food delivery application focusing on an Italian restaurant as a case study to understand how array destructuring works in practice.

Example Scenario: Consider the following restaurant object:

```
const restaurant = {
  name: "Classico Italiano",
  location: "Via Angelo Tavanti 23, Firenze, Italy",
  categories: ["Italian", "Pizzeria", "Vegetarian", "Organic"],
  starterMenu: ["Focaccia", "Bruschetta", "Garlic Bread", "Caprese Salad"],
  mainMenu: ["Pizza", "Pasta", "Risotto"],
};
```

What is Destructuring? Destructuring is a way to unpack values from arrays or properties from objects into distinct variables. It simplifies the extraction process, making the code cleaner and more concise. Array

Destructuring is an ES6 feature that allows you to unpack values from arrays or objects into separate variables. This makes it easier to extract values without writing repetitive code.

Basic Array Destructuring:

1. **Without Destructuring:** If you want to extract the first three elements of an array and assign them to variables, you would typically do:

```
const arr = [2, 3, 4];
const a = arr[0];
const b = arr[1];
const c = arr[2];
```

2. **With Destructuring:** You can achieve the same result with much less code:

```
const [x, y, z] = [2, 3, 4];
console.log(x, y, z); // Output: 2 3 4
```

   Here, `x` is assigned the first value of the array, `y` the second, and `z` the third.

Practical Example:

Let's use the `categories` array from our `restaurant` object:

```
const [first, second] = restaurant.categories;
console.log(first, second); // Output: Italian Pizzeria
```

Skipping Elements: You can skip elements in the array by leaving a gap in the destructuring pattern:

```
const [main, , secondary] = restaurant.categories;
console.log(main, secondary); // Output: Italian Vegetarian
```

Switching Variables: One of the cool tricks with destructuring is switching variables without the need for a temporary variable:

```
let [main, , secondary] = restaurant.categories;
[main, secondary] = [secondary, main];
console.log(main, secondary); // Output: Vegetarian Italian
```

Destructuring from Function Returns: You can destructure values returned from functions:

```

```

```
const order = function (starterIndex, mainIndex) {
  return [restaurant.starterMenu[starterIndex], restaurant.mainMenu[mainIndex]];
};

const [starter, mainCourse] = order(2, 0);
console.log(starter, mainCourse); // Output: Garlic Bread Pizza
```

Nested Destructuring: Destructuring can handle nested arrays as well:

```
const nested = [2, 4, [5, 6]];
const [i, , [j, k]] = nested;
console.log(i, j, k); // Output: 2 5 6
```

Default Values: If you try to destructure an array with fewer elements than expected, you can provide default values:

```
const [p = 1, q = 1, r = 1] = [8, 9];
console.log(p, q, r); // Output: 8 9 1
```

Conclusion: Array destructuring is a powerful feature in modern JavaScript that allows for cleaner and more readable code, especially when dealing with complex data structures like arrays and objects. Through the restaurant example, we've seen how destructuring simplifies the extraction of values and even makes switching variables more straightforward.

Object Destructuring in JavaScript

Object destructuring is a powerful feature in JavaScript that allows you to extract properties from objects and assign them to variables in a concise manner. Below are the key concepts covered:

1. Basic Object Destructuring:

To destructure an object, you use curly braces `{}` and specify the property names you want to extract:

```
const restaurant = {
  name: "Classico Italiano",
  location: "Via Angelo Tavanti 23, Firenze, Italy",
  categories: ["Italian", "Pizzeria", "Vegetarian", "Organic"],
  starterMenu: ["Focaccia", "Bruschetta", "Garlic Bread", "Caprese Salad"],
  mainMenu: ["Pizza", "Pasta", "Risotto"],
  openingHours: {
    mon: { open: 12, close: 22 },
    fri: { open: 11, close: 23 },
    sat: { open: 0, close: 24 },
  },
};
```

```
// Destructuring name, categories, and openingHours from the restaurant object
const { name, categories, openingHours } = restaurant;
console.log(name); // 'Classico Italiano'
console.log(categories); // ['Italian', 'Pizzeria', 'Vegetarian', 'Organic']
console.log(openingHours); // { mon: { open: 12, close: 22 }, fri: { open: 11,
close: 23 }, sat: { open: 0, close: 24 } }
```

2. Renaming Variables:

You can rename the variables while destructuring by using a colon `:`:

```
const {
  name: restaurantName,
  openingHours: hours,
  categories: tags,
} = restaurant;
console.log(restaurantName); // 'Classico Italiano'
console.log(hours); // { mon: { open: 12, close: 22 }, fri: { open: 11, close: 23
}, sat: { open: 0, close: 24 } }
console.log(tags); // ['Italian', 'Pizzeria', 'Vegetarian', 'Organic']
```

3. Setting Default Values:

If a property does not exist in the object, you can set a default value:

```
const { menu = [], starterMenu: starters = [] } = restaurant;
console.log(menu); // []
console.log(starters); // ['Focaccia', 'Bruschetta', 'Garlic Bread', 'Caprese
Salad']
```

4. Mutating Variables:

If you need to destructure into existing variables, wrap the destructuring assignment in parentheses:

```
let a = 111;
let b = 999;

const obj = { a: 23, b: 7, c: 14 };

// Mutating variables
({ a, b } = obj);
console.log(a); // 23
console.log(b); // 7
```

5. Nested Destructuring:

You can destructure nested objects by following the structure of the object:

```
const {
  fri: { open, close },
} = openingHours;
console.log(open); // 11
console.log(close); // 23
```

6. Practical Application with Function Parameters:

Destructuring is often used in function parameters to make code more readable:

```
const orderDelivery = function ({
  starterIndex = 1,
  mainIndex = 0,
  time = "20:00",
  address,
}) {
  console.log(
    `Order received! ${restaurant.starterMenu[starterIndex]} and
${restaurant.mainMenu[mainIndex]} will be delivered to ${address} at ${time}`
  );
};

// Calling the function with an object as an argument
orderDelivery({
  time: "22:30",
  address: "Via del Sole, 21",
  mainIndex: 2,
  starterIndex: 2,
});

// Using default values
orderDelivery({
  address: "Via del Sole, 21",
  starterIndex: 1,
});
```

- **Explanation**: By passing an object to the function, you can easily manage multiple parameters without worrying about their order. Default values ensure the function works even if some parameters are not provided.

Key Takeaways:

- **Destructuring** simplifies extracting properties from objects.
- **Renaming** variables and setting **default values** make the code more flexible.

- Destructuring is particularly useful in functions, especially with API data or complex parameter lists.

This method reduces the risk of errors, making the code cleaner and easier to maintain, especially when dealing with complex data structures or third-party data.

Spread Operator in JavaScript

The spread operator (`...`) is a powerful feature in JavaScript, introduced in ES6, that allows you to expand iterable elements (like arrays, strings, maps, and sets) into individual elements. This operator simplifies operations such as copying arrays, combining arrays, passing arguments to functions, and even creating new objects by merging properties.

Basic Usage

Imagine you have an array `arr = [7, 8, 9]` and you want to create a new array with some additional elements. Without the spread operator, you might manually add the elements or loop through the array. With the spread operator, this becomes much simpler:

```
const arr = [7, 8, 9];
const newArr = [1, 2, ...arr];
console.log(newArr); // Output: [1, 2, 7, 8, 9]
```

Here, `...arr` expands the elements of `arr` into the new array.

Use Cases

1. **Expanding Elements in Function Calls**: The spread operator can be used to pass the elements of an array as individual arguments to a function.

   ```
   const newArr = [1, 2, 7, 8, 9];
   console.log(...newArr); // Output: 1 2 7 8 9
   ```

2. **Creating a New Array**: It can also be used to create a new array by adding elements to the beginning or end of an existing array.

   ```
   const restaurant = {
     mainMenu: ["Pizza", "Pasta", "Risotto"],
   };
   const newMenu = [...restaurant.mainMenu, "Gnocci"];
   console.log(newMenu); // Output: ['Pizza', 'Pasta', 'Risotto', 'Gnocci']
   ```

3. **Shallow Copy of an Array**: You can create a shallow copy of an array, which means you duplicate the array without affecting the original one.

```
const mainMenuCopy = [...restaurant.mainMenu];
console.log(mainMenuCopy); // Output: ['Pizza', 'Pasta', 'Risotto']
```

4. **Merging Arrays**: The spread operator allows you to easily merge multiple arrays into one.

```
const starterMenu = ["Focaccia", "Bruschetta", "Garlic Bread"];
const menu = [...starterMenu, ...restaurant.mainMenu];
console.log(menu); // Output: ['Focaccia', 'Bruschetta', 'Garlic Bread',
'Pizza', 'Pasta', 'Risotto']
```

5. **Spread Operator with Strings**: Since strings are iterables, the spread operator can split a string into individual characters.

```
const str = "Hello";
const letters = [...str, " ", "W", "o", "r", "l", "d"];
console.log(letters); // Output: ['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o',
'r', 'l', 'd']
```

Spread Operator with Objects

While the spread operator was originally designed for arrays, it can also be used with objects since ES2018. This allows you to easily copy objects or merge multiple objects together.

```
const restaurant = {
  name: "Classico Italiano",
  location: "Rome, Italy",
  categories: ["Italian", "Pizzeria", "Vegetarian", "Organic"],
};
const newRestaurant = { ...restaurant, founder: "Giuseppe", foundedIn: 1998 };
console.log(newRestaurant);
// Output: {
//   name: 'Classico Italiano',
//   location: 'Rome, Italy',
//   categories: ['Italian', 'Pizzeria', 'Vegetarian', 'Organic'],
//   founder: 'Giuseppe',
//   foundedIn: 1998
// }
```

Differences from Destructuring

- **Destructuring** allows you to unpack elements from arrays or properties from objects into distinct variables.
- **Spread Operator** expands elements from arrays or strings, or properties from objects, without creating new variables.

Important Notes

- The spread operator can only be used where values are separated by commas, such as in array literals or function calls.
- It does not work directly within template literals or any other context that doesn't expect multiple comma-separated values.

The spread operator is a modern and efficient way to handle arrays and objects in JavaScript, making your code cleaner and easier to manage.

Rest Pattern and Rest Parameters in JavaScript

Understanding the Rest Pattern:

The **rest pattern** in JavaScript is used to collect multiple elements into an array, essentially condensing them. While it shares the same syntax as the spread operator (three dots: `...`), it serves the opposite purpose. The spread operator is used to expand an array into individual elements, while the rest pattern is used to pack elements into an array.

Key Concepts:

1. **Spread Operator**:
    - **Purpose**: To expand an array or object into individual elements or properties.
    - **Use Cases**:
        - Creating new arrays or objects by spreading elements/properties from an existing array/object.
        - Passing multiple arguments to a function.
2. **Rest Pattern**:
    - **Purpose**: To collect multiple elements and condense them into an array.
    - **Use Cases**:
        - Destructuring arrays or objects to capture remaining elements or properties into a new array or object.
        - Function parameters to accept an arbitrary number of arguments.

Rest Pattern in Arrays:

The rest pattern is used on the left-hand side of an assignment during array destructuring. It collects the "rest" of the elements that are not explicitly assigned to individual variables.

```
const [a, b, ...others] = [1, 2, 3, 4, 5];
console.log(a); // 1
console.log(b); // 2
console.log(others); // [3, 4, 5]
```

- Here, `a` and `b` are assigned the first two elements, and the rest of the elements (`3, 4, 5`) are collected into the `others` array.

Rest Pattern in Objects:

Similar to arrays, the rest pattern can be used in object destructuring. It collects remaining properties into a new object.

```
const { sat, ...weekdays } = restaurant.openingHours;
console.log(sat); // Saturday opening hours
console.log(weekdays); // Object containing Thursday and Friday hours
```

- The `sat` variable captures the `Saturday` property, and the rest (Thursday and Friday) are collected into the `weekdays` object.

Rest Parameters in Functions:

The rest pattern can also be used in function parameters, known as **rest parameters**. It allows functions to accept an arbitrary number of arguments, which are then collected into an array.

```
function add(...numbers) {
  let sum = 0;
  for (const num of numbers) sum += num;
  return sum;
}

console.log(add(2, 3)); // 5
console.log(add(5, 3, 7, 2)); // 17
```

- The `add` function can take any number of arguments. All arguments are collected into the `numbers` array and then summed up.

Combining Spread and Rest:

You can use the spread operator to pass an array to a function with rest parameters. The spread operator expands the array into individual elements, and the rest parameters collect them back into an array within the function.

```
const numbers = [23, 5, 7];
console.log(add(...numbers)); // 35
```

Important Notes:

- **Rest Pattern in Destructuring**:

    - The rest element must be the last element in the destructuring assignment.
    - There can only be one rest element in any destructuring assignment.

- **Rest Parameters in Functions**:

    - Rest parameters must be the last parameter in the function signature.
    - There can only be one rest parameter in a function.

Recap:

- **Spread Operator**: Expands an array/object into individual elements/properties.
- **Rest Pattern**: Collects remaining elements/properties into an array/object.
- **Use Cases**:
    - **Spread**: Used in array/object literals and function calls.
    - **Rest**: Used in destructuring and function parameters. This lesson delves into the concept of "short circuiting" with logical operators, focusing on the AND (`&&`) and OR (`||`) operators in JavaScript. Let's break it down:

Short Circuiting with OR (`||`) Operator

1. **Basic Concept**:

    - The OR operator returns the first truthy value it encounters.
    - If the first operand is truthy, the evaluation stops, and the first operand is returned.
    - If the first operand is falsy, the OR operator evaluates the second operand and returns it.

2. **Examples**:

```
console.log(3 || "Jonas"); // 3
console.log("" || "Jonas"); // 'Jonas'
console.log(true || 0); // true
console.log(undefined || null); // null
```

    - `3 || 'Jonas'` returns `3` because `3` is truthy.
    - `'' || 'Jonas'` returns `'Jonas'` because `''` is falsy.

3. **Chaining OR Operator**:

    - When chaining multiple OR operations, the evaluation continues until a truthy value is found.
    - Example:

```
console.log(undefined || 0 || "" || "hello" || 23 || null); // 'hello'
```

        - `hello` is the first truthy value, so it's returned.

Practical Application of OR Operator:

- **Setting Default Values**:

```
const guests = restaurant.numGuests || 10;
```

  - If `restaurant.numGuests` is undefined or falsy, it defaults to `10`.

- **Limitation**:
  - If `numGuests` is `0`, the above method will incorrectly assign `10` because `0` is falsy. A solution to this issue will be explored later.

Short Circuiting with AND (&&) Operator

1. **Basic Concept**:
   - The AND operator returns the first falsy value it encounters.
   - If the first operand is falsy, the evaluation stops, and the first operand is returned.
   - If the first operand is truthy, the AND operator evaluates the second operand and returns it.
2. **Examples**:

```
console.log(0 && "Jonas"); // 0
console.log(7 && "Jonas"); // 'Jonas'
console.log("Hello" && 23 && null && "Jonas"); // null
```

   - `0 && 'Jonas'` returns `0` because `0` is falsy.
   - `'Hello' && 23 && null && 'Jonas'` returns `null` because `null` is the first falsy value.

Practical Application of AND Operator:

- **Conditional Execution**:

```
restaurant.orderPizza && restaurant.orderPizza("mushrooms", "spinach");
```

  - If `restaurant.orderPizza` exists, the function is executed; otherwise, nothing happens.

Summary:

- The OR (||) operator returns the first truthy value or the last falsy value.
- The AND (&&) operator returns the first falsy value or the last truthy value.
- These operators can be used for setting default values or conditional execution, but with care to avoid potential issues like incorrectly handling `0` as a falsy value. The nullish coalescing operator (??) is indeed a handy tool for handling default values in situations where you want to avoid false positives from other falsy values like `0` or `''`. Here's a quick recap:

Nullish Coalescing Operator (??)

Purpose: To provide a default value when the left-hand side value is `null` or `undefined`, but not for other falsy values like `0` or `''`.

How It Works:

- **When Left-Hand Side is `null` or `undefined`**: The operator returns the right-hand side value.
- **When Left-Hand Side is Any Other Value**: The operator returns the left-hand side value, regardless of whether it is falsy (e.g., `0`, `''`).

Examples

1. **Using Nullish Coalescing Operator:**

   ```
   const numGuests = 0;
   const guests = numGuests ?? 10; // guests will be 0
   ```

   In this case, `numGuests` is `0`, which is neither `null` nor `undefined`, so `guests` is assigned the value `0`.

2. **Default Value Case:**

   ```
   const numGuests = undefined;
   const guests = numGuests ?? 10; // guests will be 10
   ```

   Here, `numGuests` is `undefined`, so `guests` is assigned the default value `10`.

3. **Combining with Other Values:**

   ```
   const name = "";
   const displayName = name ?? "Anonymous"; // displayName will be ''
   ```

   Since `name` is an empty string (which is falsy but not nullish), `displayName` is assigned `''`.

This operator can be particularly useful when you want to ensure that defaults are applied only for `null` and `undefined` values, avoiding the unintended side effects of treating other falsy values as if they were missing or undefined.

Logical Assignment Operators

The logical assignment operators introduced in ES2021 provide a more concise way to perform common assignments that involve logical operations. Here's a summary of each operator:

1. **Logical OR Assignment (`||=`)**
   - **Syntax**: `x ||= y`
   - **Description**: Assigns `y` to `x` if `x` is falsy.
   - **Example**:

```
let guests = 0;
guests ||= 10; // guests is 0, which is falsy, so guests becomes 10
console.log(guests); // Output: 10
```

2. **Logical AND Assignment (`&&=`)**

   - **Syntax**: `x &&= y`
   - **Description**: Assigns `y` to `x` if `x` is truthy.
   - **Example**:

```
let owner = "Giovanni Rossi";
owner &&= "Anonymous"; // owner is truthy, so owner becomes 'Anonymous'
console.log(owner); // Output: Anonymous
```

3. **Logical Nullish Assignment (`??=`)**

   - **Syntax**: `x ??= y`
   - **Description**: Assigns `y` to `x` if `x` is nullish (`null` or `undefined`).
   - **Example**:

```
let guests = undefined;
guests ??= 10; // guests is undefined, so guests becomes 10
console.log(guests); // Output: 10
```

Detailed Examples

Logical OR Assignment (`||=`)

```
let restaurant1 = { name: "La Piazza", numGuests: 20 };
let restaurant2 = { name: "Bella Italia" };

// Setting default number of guests
restaurant2.numGuests ||= 10;
console.log(restaurant2.numGuests); // Output: 10

// Number of guests is already set in restaurant1
restaurant1.numGuests ||= 10;
console.log(restaurant1.numGuests); // Output: 20
```

Here, `restaurant2.numGuests` was undefined, so it was assigned the default value `10`. In contrast, `restaurant1.numGuests` was already `20`, so it remained unchanged.

Logical AND Assignment (`&&=`)

```
let restaurant1 = { name: "La Piazza", owner: "Giovanni Rossi" };
let restaurant2 = { name: "Bella Italia" };

// Anonymizing the owner
restaurant1.owner &&= "Anonymous";
console.log(restaurant1.owner); // Output: Anonymous

// No owner to anonymize
restaurant2.owner &&= "Anonymous";
console.log(restaurant2.owner); // Output: undefined (unchanged)
```

In this case, `restaurant1.owner` was truthy, so it was replaced with `'Anonymous'`. For `restaurant2`, since there was no `owner` property, it remained unchanged.

Logical Nullish Assignment (`??=`)

```
let restaurant1 = { name: "La Piazza", numGuests: 20 };
let restaurant2 = { name: "Bella Italia" };

// Setting default number of guests only if it's nullish
restaurant2.numGuests ??= 10;
console.log(restaurant2.numGuests); // Output: 10

// Number of guests is already set in restaurant1
restaurant1.numGuests ??= 10;
console.log(restaurant1.numGuests); // Output: 20
```

Here, `restaurant2.numGuests` was undefined, so it was assigned `10`. `restaurant1.numGuests` was already `20`, so it was not changed.

These operators simplify the code by reducing redundancy and making logical assignments more readable and concise. The `for-of` loop introduced in ES6 is a modern, simplified way to iterate over arrays and other iterable objects. It eliminates the need for manual index management, making loops more readable and easier to work with. Here's a detailed breakdown of how it works and its features:

`for-of` Loop

Basic Usage

The `for-of` loop allows you to iterate directly over the values of an array or iterable object. The syntax is:

```
for (const item of iterable) {
  // code to execute for each item
}
```

- `item`: A variable that holds the current value from the iterable.

- `iterable`: The array or iterable object you want to loop through.

Example:

```
const menu = ["Starter", "Main Course", "Dessert"];

for (const item of menu) {
  console.log(item);
}

// Output:
// Starter
// Main Course
// Dessert
```

In this example, `item` represents each element of the `menu` array during each iteration.

Accessing Index with `for-of`

The `for-of` loop does not provide the index directly, but you can use the `entries()` method to get both index and element.

Example:

```
const menu = ["Starter", "Main Course", "Dessert"];

for (const [index, item] of menu.entries()) {
  console.log(`Item ${index + 1}: ${item}`);
}

// Output:
// Item 1: Starter
// Item 2: Main Course
// Item 3: Dessert
```

- `menu.entries()`: Returns an iterator object that contains arrays of `[index, element]` pairs.
- **Destructuring**: `[index, item]` is used to extract index and element from the `[index, element]` array.

Using `continue` and `break`

The `for-of` loop supports `continue` and `break` statements to control the flow of the loop.

Example:

```
const menu = ["Starter", "Main Course", "Dessert"];
```

```
for (const item of menu) {
  if (item === "Main Course") {
    continue; // Skip 'Main Course'
  }
  console.log(item);
}

// Output:
// Starter
// Dessert
```

Example with `break`:

```
const menu = ["Starter", "Main Course", "Dessert"];

for (const item of menu) {
  if (item === "Main Course") {
    break; // Exit loop when 'Main Course' is encountered
  }
  console.log(item);
}

// Output:
// Starter
```

Summary

- **Simpler Syntax**: The `for-of` loop simplifies iteration by directly accessing array elements, avoiding manual index management.
- **Readable Code**: Reduces boilerplate code and makes loops more readable.
- **Entries Method**: Use `entries()` method to access both index and element if needed.
- **Supports Control Flow**: Allows use of `continue` and `break` for loop control.

The `for-of` loop is especially useful when you need to iterate over arrays or other iterable objects without the complexity of managing indices and conditions. ES6 introduced several enhancements to object literals that simplify and streamline how you write and manage objects in JavaScript. Here's a detailed look at these enhancements:

1. **Property Shorthand**

Before ES6, when you wanted to add a property to an object where the property name and the variable name were the same, you had to repeat the variable name:

Before ES6:

```
const openingHours = {
  weekday: "9am - 5pm",
  weekend: "10am - 4pm",
```

```
};

const restaurant = {
  name: "Cafe Good Vibes",
  openingHours: openingHours, // Property name and variable name are the same
};
```

With ES6 Property Shorthand:

```
const openingHours = {
  weekday: "9am - 5pm",
  weekend: "10am - 4pm",
};

const restaurant = {
  name: "Cafe Good Vibes",
  openingHours, // Shortened syntax
};
```

- **Benefit**: Reduces redundancy by automatically using the variable name as the property name.

2. **Method Shorthand**

Before ES6, methods in objects were defined using function expressions:

Before ES6:

```
const restaurant = {
  name: "Cafe Good Vibes",
  getOpeningHours: function () {
    return "9am - 5pm";
  },
};
```

With ES6 Method Shorthand:

```
const restaurant = {
  name: "Cafe Good Vibes",
  getOpeningHours() {
    return "9am - 5pm";
  },
};
```

- **Benefit**: Simplifies the syntax for defining methods within objects.

3. **Computed Property Names**

ES6 allows you to use expressions inside square brackets `[]` to define property names dynamically:

Before ES6:

```
const weekdays = [
  "Monday",
  "Tuesday",
  "Wednesday",
  "Thursday",
  "Friday",
  "Saturday",
  "Sunday",
];

const openingHours = {
  Thursday: "9am - 5pm",
  Friday: "9am - 5pm",
};
```

With ES6 Computed Property Names:

```
const weekdays = [
  "Monday",
  "Tuesday",
  "Wednesday",
  "Thursday",
  "Friday",
  "Saturday",
  "Sunday",
];

const openingHours = {
  [weekdays[3]]: "9am - 5pm", // Thursday
  [weekdays[4]]: "9am - 5pm", // Friday
};
```

Using Expressions:

```
const day = 2 + 2; // Example expression
const openingHours = {
  [`Day ${day}`]: "9am - 5pm", // Day 4
};
```

- **Benefit**: Allows property names to be calculated at runtime, making object definitions more dynamic and flexible.

Summary of Enhancements

1. **Property Shorthand**: Use the variable name directly as the property name.
2. **Method Shorthand**: Simplify the syntax for defining methods.
3. **Computed Property Names**: Dynamically create property names using expressions within square brackets.

These enhancements make working with object literals more efficient and intuitive, reducing boilerplate code and improving readability. Optional chaining, introduced in ES2020, is a powerful feature that allows you to safely access deeply nested properties of an object without having to check each level for existence. Here's a detailed overview of how optional chaining works and its practical uses:

Understanding Optional Chaining

Basic Concept

Optional chaining is used to safely access nested properties of an object. It short-circuits and returns `undefined` if any part of the chain is `null` or `undefined`, preventing runtime errors.

Syntax

The optional chaining operator is `?.`. It can be used with dot notation and brackets:

- **Dot Notation**: `object?.property`
- **Bracket Notation**: `object?.[property]`
- **Function Calls**: `object?.method()`

Examples

1. **Accessing Nested Properties:**

```
const restaurant = {
  name: "Cafe Good Vibes",
  openingHours: {
    Monday: {
      open: "9am",
      close: "5pm",
    },
  },
};

const mondayOpen = restaurant.openingHours?.Monday?.open; // '9am'
const sundayOpen = restaurant.openingHours?.Sunday?.open; // undefined
```

   - If `openingHours` or `Monday` is `null` or `undefined`, `undefined` will be returned instead of throwing an error.

2. **Calling Methods Safely:**

```
const restaurant = {
  name: "Cafe Good Vibes",
  order: (item) => `Ordering ${item}`,
};

const result = restaurant.order?.("coffee"); // 'Ordering coffee'
const resultInvalid = restaurant.orderInvalid?.("coffee"); // undefined
```

- If `orderInvalid` doesn't exist, it will return `undefined` instead of throwing an error.

3. **Working with Arrays:**

```
const users = [{ name: "Jonas" }];

const firstUserName = users[0]?.name; // 'Jonas'
const secondUserName = users[1]?.name; // undefined
```

- If `users[1]` is `undefined`, `undefined` will be returned instead of causing an error.

4. **Using with Default Values:**

```
const restaurant = {
  openingHours: {
    Monday: {
      open: 12,
    },
  },
};

const mondayOpen = restaurant.openingHours?.Monday?.open ?? "Closed"; // 12
const sundayOpen = restaurant.openingHours?.Sunday?.open ?? "Closed"; //
'Closed'
```

- Combined with the nullish coalescing operator (`??`), optional chaining can handle cases where values might be `null` or `undefined`.

Practical Use Case

Optional chaining is especially useful in situations where you're dealing with data from APIs or dynamic sources where properties might be missing or undefined:

1. **Accessing Data from APIs:**

```
const response = await fetch("api/restaurant");
const data = await response.json();
```

```
const mondayHours = data?.openingHours?.Monday?.open ?? "Not available";
```

- This ensures that even if some properties are missing, your code will not break and will return a fallback value.

2. **Looping Through Arrays:**

```
const days = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday"];

days.forEach((day) => {
  const hours = restaurant.openingHours?.[day]?.open ?? "Closed";
  console.log(`${day}: Open at ${hours}`);
});
```

- This handles cases where some days might not have opening hours without causing errors.

Conclusion

Optional chaining is a powerful feature that simplifies accessing deeply nested properties and methods, improving code readability and reducing the risk of runtime errors. Combined with the nullish coalescing operator, it provides a robust way to handle missing or undefined values gracefully. Great walkthrough! Here's a summary of the key points on iterating over object properties in JavaScript:

Iterating Over Object Properties

1. **Looping Over Property Names (Keys)**:
   - Use `Object.keys()` to get an array of property names (keys) of an object.
   - Example:

```
const openingHours = {
  Thursday: { open: 12, close: 22 },
  Friday: { open: 12, close: 22 },
  Saturday: { open: 0, close: 24 },
};

const days = Object.keys(openingHours);
console.log(days); // ["Thursday", "Friday", "Saturday"]
```

2. **Looping Over Property Values**:
   - Use `Object.values()` to get an array of property values of an object.
   - Example:

```

```

```
const values = Object.values(openingHours);
console.log(values); // [{ open: 12, close: 22 }, { open: 12, close: 22
}, { open: 0, close: 24 }]
```

3. **Looping Over Both Keys and Values**:
    - Use `Object.entries()` to get an array of key-value pairs from an object.
    - Example:

```
const entries = Object.entries(openingHours);
console.log(entries); // [["Thursday", { open: 12, close: 22 }],
["Friday", { open: 12, close: 22 }], ["Saturday", { open: 0, close: 24
}]]
```

    - To loop over these entries:

```
for (const [day, { open, close }] of Object.entries(openingHours)) {
  console.log(`On ${day}, we open at ${open} and close at ${close}.`);
}
// Output:
// On Thursday, we open at 12 and close at 22.
// On Friday, we open at 12 and close at 22.
// On Saturday, we open at 0 and close at 24.
```

Key Points:

- **Object.keys()**: Returns an array of the object's own enumerable property names (keys).
- **Object.values()**: Returns an array of the object's own enumerable property values.
- **Object.entries()**: Returns an array of the object's own enumerable property `[key, value]` pairs.

Using these methods, you can efficiently iterate over object properties based on what you need to access—keys, values, or both. You've provided a comprehensive overview of JavaScript Sets. Here's a concise summary of the key points:

JavaScript Sets

Overview:

- Sets are collections of unique values, introduced in ES6.
- They eliminate duplicate values and are useful for scenarios where uniqueness is important.

Creating a Set:

- Initialize a set with `new Set()`, passing an iterable (like an array) if desired.

```

```

```
const ordersSet = new Set([
  "pasta",
  "pizza",
  "pizza",
  "risotto",
  "pasta",
  "pizza",
]);
console.log(ordersSet); // Set { 'pasta', 'pizza', 'risotto' }
```

Working with Sets:

1. **Size:**
   - `Set.prototype.size` returns the number of unique elements in the set.

     ```
     console.log(ordersSet.size); // 3
     ```

2. **Checking for Existence:**
   - `Set.prototype.has(value)` checks if a value is in the set.

     ```
     console.log(ordersSet.has("pizza")); // true
     console.log(ordersSet.has("bread")); // false
     ```

3. **Adding Elements:**
   - `Set.prototype.add(value)` adds a new element to the set. Duplicates are ignored.

     ```
     ordersSet.add("garlic bread");
     console.log(ordersSet); // Set { 'pasta', 'pizza', 'risotto', 'garlic
     bread' }
     ```

4. **Deleting Elements:**
   - `Set.prototype.delete(value)` removes an element from the set.

     ```
     ordersSet.delete("risotto");
     console.log(ordersSet); // Set { 'pasta', 'pizza', 'garlic bread' }
     ```

5. **Clearing the Set:**
   - `Set.prototype.clear()` removes all elements from the set.

```
ordersSet.clear();
console.log(ordersSet); // Set {}
```

6. **Looping Over a Set:**
    - Use `for...of` to iterate over the elements.

```
for (const order of ordersSet) {
  console.log(order);
}
```

Converting a Set to an Array:

- Use the spread operator to convert a set to an array.

```
const staff = ["waiter", "chef", "waiter", "manager", "chef", "waiter"];
const uniqueStaff = [...new Set(staff)];
console.log(uniqueStaff); // ['waiter', 'chef', 'manager']
```

Use Cases:

- **Removing Duplicates:** Sets are commonly used to remove duplicates from arrays.
- **Count Unique Values:** You can use `Set.prototype.size` to quickly find the number of unique values.

Important Notes:

- Sets do not maintain element order, and they do not support indexing like arrays.
- Use sets when uniqueness is required; otherwise, arrays are generally more suitable.

Sets provide an efficient way to handle collections of unique values, and they can be particularly useful for eliminating duplicates and checking for existence. Here's a detailed summary of the lecture on Maps in JavaScript:

Overview of Maps

- **Definition**: A Map is a data structure that maps values to keys, similar to objects but with some important differences.
- **Key Differences from Objects**:
    - **Key Types**: Maps can have keys of any type (e.g., objects, arrays, primitive values), while objects only use strings or symbols as keys.
    - **Order of Keys**: Maps remember the insertion order of keys, which is not guaranteed with objects.

Creating and Using Maps

1. **Creating a Map**:

- Use the `new Map()` constructor to create an empty Map.
- Example:

```
let rest = new Map();
```

2. **Adding Elements**:

- Use the `set(key, value)` method to add or update elements.
- Example:

```
rest.set("name", "Classico Italiano");
rest.set(1, "Firenze, Italy");
rest.set(2, "Lisbon, Portugal");
```

3. **Chaining Set Operations**:

- `set` returns the Map itself, allowing method chaining.
- Example:

```
rest
  .set("categories", ["Italian", "Pasta"])
  .set("open", 11)
  .set("close", 23)
  .set(true, "We are open")
  .set(false, "We are closed");
```

4. **Retrieving Values**:

- Use the `get(key)` method to retrieve values based on keys.
- Example:

```
console.log(rest.get("name")); // 'Classico Italiano'
console.log(rest.get(true)); // 'We are open'
```

5. **Using Maps with Conditions**:

- Example of using boolean keys for conditional logic:

```
let currentTime = 21; // 9:00 PM
let status =
  currentTime > rest.get("open") && currentTime < rest.get("close")
    ? rest.get(true)
    : rest.get(false);
console.log(status); // 'We are open'
```

Additional Methods

1. **Checking Existence**:
   - Use `has(key)` to check if a key exists.
   - Example:

```
console.log(rest.has("categories")); // true
```

2. **Deleting Elements**:
   - Use `delete(key)` to remove a key-value pair.
   - Example:

```
rest.delete(2); // Removes Lisbon, Portugal
```

3. **Getting Size**:
   - Use `size` to get the number of elements in the Map.
   - Example:

```
console.log(rest.size); // Number of entries
```

4. **Clearing the Map**:
   - Use `clear()` to remove all entries.
   - Example:

```
rest.clear(); // Removes all entries
```

Special Key Types

- **Using Objects as Keys**:
  - Maps can use objects as keys. Keys must be the exact same object reference.
  - Example:

```
let keyArray = [1, 2];
rest.set(keyArray, "Test");
console.log(rest.get(keyArray)); // 'Test'
```

- **Using DOM Elements as Keys**:
  - Maps can also use DOM elements as keys.
  - Example:

```
let heading = document.querySelector("h1");
rest.set(heading, "Heading Element");
```

Conclusion

- Maps are useful for scenarios where keys are not just strings and where maintaining the order of keys is important.
- For ordered data or frequent updates, use arrays.
- Maps provide a robust API for managing key-value pairs with various key types.

Feel free to ask if you need more details or examples! Great, let's dive into the key points from the video on Maps:

Creating Maps

1. **Initialization with Arrays**:
   - You can initialize a `Map` using an array of arrays. Each inner array represents a key-value pair.
   - Example:

```
const question = new Map([
  ["What is the best programming language in the world?",
"JavaScript"],
  [1, "C"],
  [2, "Java"],
  [3, "JavaScript"],
  ["correct", 3],
  [true, "Correct! 🎉"],
  [false, "Try again."],
]);
```

2. **Initialization with `object.entries`**:
   - Convert an object to a `Map` using `Object.entries()`.
   - Example:

```
const openingHours = {
  monday: "9am - 5pm",
  tuesday: "10am - 6pm",
  // ...other days
};
const hoursMap = new Map(Object.entries(openingHours));
```

Accessing Data

1. **Using the `get` Method**:
   - Retrieve values by passing the key to `get()`.
   - Example:

```
console.log(question.get(1)); // Output: 'C'
```

2. **Handling Boolean Keys**:
   - Maps can use booleans as keys, which can be useful for conditions.
   - Example:

```
const currentTime = 21; // 9 PM
const status =
  currentTime > question.get("open") &&
  currentTime < question.get("close");
console.log(question.get(status)); // Output: 'Correct! 🎉' or 'Try
again.'
```

Iterating Over Maps

1. **Using `for...of` Loop**:
   - Iterate over `Map` entries using a `for...of` loop.
   - Example:

```
for (const [key, value] of question) {
  console.log(key, value);
}
```

Map Methods and Properties

1. `has(key)`:
   - Check if a key exists in the map.
   - Example:

```
console.log(question.has(2)); // Output: true
```

2. `delete(key)`:
   - Remove a key-value pair from the map.

- Example:

```
question.delete(2);
```

3. `size`:
   - Get the number of key-value pairs.
   - Example:

```
console.log(question.size); // Output: Number of pairs
```

4. `clear()`:
   - Remove all entries from the map.
   - Example:

```
question.clear();
```

Converting Maps

1. **Map to Array**:
   - Convert a `Map` to an array of arrays.
   - Example:

```
const mapArray = [...question];
```

2. **Array of Keys and Values**:
   - Use `keys()` and `values()` methods for iterators, which can be converted to arrays if needed.
   - Example:

```
const keysArray = [...question.keys()];
const valuesArray = [...question.values()];
```

When to Use Maps vs Objects

- **Use Maps**:
  - When you need keys of any type (including objects).
  - When you need to maintain the order of elements.
  - When you need to frequently add or remove key-value pairs.

- **Use Objects**:
    - When you need simple key-value pairs with string keys.
    - When the structure of the keys is fixed and does not need to change dynamically.

This concludes the lecture on Maps. If you have any questions or need further clarification on any of the concepts, feel free to ask! Here's a summary of the key points from the lecture on choosing between JavaScript data structures:

Sources of Data

1. **Within the Program**:
    - Data hardcoded into the source code (e.g., status messages).
2. **From the User Interface**:
    - Data input by users or data present in the DOM (e.g., tasks in a to-do app).
3. **From External Sources**:
    - Data fetched from web APIs (e.g., weather data, movie data).

JavaScript Data Structures

1. Arrays vs Sets:

- **Arrays**:
    - Use for ordered lists of values.
    - Allows duplicate values.
    - Provides many methods for manipulating data (e.g., `map()`, `filter()`, `reduce()`).
- **Sets**:
    - Use for collections of unique values.
    - Provides efficient operations for checking existence, adding, and removing items.
    - Ideal for removing duplicates from arrays.

2. Objects vs Maps:

- **Objects**:
    - Traditional way to store key-value pairs.
    - Key limitations: Keys must be strings or symbols, and performance can be slower for certain operations.
    - Best used for storing methods (functions) and JSON data.
- **Maps**:
    - Designed for key-value pairs with any data type as keys.
    - Offers better performance for frequent additions, deletions, and lookups.
    - Provides straightforward iteration over entries.
    - Ideal for scenarios where keys are not strings or symbols.

When to Use Each Data Structure

- **Use Arrays**:
    - When you need to store values in order.
    - When duplicates are allowed.
    - When you need to manipulate data with built-in array methods.
- **Use Sets**:
    - When you need to ensure all values are unique.
    - When performance is critical, especially for operations like searching and deleting.
- **Use Objects**:
    - For simple key-value pairs where keys are strings.
    - When you need methods (functions) as values.
    - When working with JSON data.
- **Use Maps**:
    - For key-value pairs where keys can be of any type.
    - When you need efficient performance for frequent operations.
    - When you need to iterate over entries or compute the size of the structure.

Additional Notes

- **WeakSets and WeakMaps**:
    - These are variations of Sets and Maps that allow garbage collection of keys.
    - Not covered in detail here but useful in specific scenarios.
- **Other Data Structures**:
    - Examples include stacks, queues, linked lists, trees, and hash tables.
    - These are not built into JavaScript but are important in programming.

Conclusion

- **Arrays** and **Objects** are still widely used and well-understood.
- **Sets** and **Maps** offer enhanced functionality and performance for specific use cases and should be considered when appropriate.

Feel free to ask if you have any questions about these data structures or need further clarification on any points! **WeakSets** and **WeakMaps** are specialized versions of Sets and Maps in JavaScript, designed to handle certain use cases with memory management in mind. Here's a breakdown of their features and differences:

WeakSets

- **Purpose**: A `WeakSet` is a collection of objects where the objects are held weakly, meaning they do not prevent garbage collection.
- **Key Features**:

    - **Weak Reference**: The objects in a `WeakSet` are weakly referenced. If there are no other references to an object stored in a `WeakSet`, it can be garbage collected.
    - **No Iteration**: `WeakSet` does not support iteration. You cannot use methods like `.forEach()` or `for...of` to iterate over its elements.
    - **No Size Property**: `WeakSet` does not have a `.size` property. There is no direct way to determine the number of elements.
    - **Operations**: The main operations supported are `.add()`, `.delete()`, and `.has()`. The `.add()` method adds an object, `.delete()` removes an object, and `.has()` checks for the presence of an object.

- **Use Cases**:

    - **Memory Management**: Useful when you need to keep track of objects without preventing them from being garbage collected. For example, keeping track of objects in a memory-sensitive application.

```
const weakSet = new WeakSet();
const obj = {};

// Add object to WeakSet
weakSet.add(obj);

// Check if object is in WeakSet
console.log(weakSet.has(obj)); // true

// Remove object from WeakSet
weakSet.delete(obj);

// Check if object is in WeakSet
console.log(weakSet.has(obj)); // false
```

WeakMaps

- **Purpose**: A `WeakMap` is a collection of key-value pairs where the keys are objects and are held weakly.

- **Key Features**:

    - **Weak Reference**: Keys in a `WeakMap` are weakly referenced. If there are no other references to a key, it can be garbage collected, and the corresponding entry will be removed.
    - **No Iteration**: `WeakMap` does not support iteration or methods like `.forEach()` or `for...of` to iterate over its entries.
    - **No Size Property**: `WeakMap` does not have a `.size` property, so you cannot directly determine the number of entries.
    - **Operations**: The main operations supported are `.set()`, `.get()`, `.has()`, and `.delete()`. The `.set()` method adds or updates a key-value pair, `.get()` retrieves a value by key, `.has()` checks if a key exists, and `.delete()` removes a key-value pair.

- **Use Cases**:

- **Metadata Storage**: Useful for storing private data or metadata associated with objects without modifying the objects themselves. For example, adding metadata to DOM elements without polluting their properties.

```
const weakMap = new WeakMap();
const key = {};

// Set value associated with key
weakMap.set(key, "value");

// Get value associated with key
console.log(weakMap.get(key)); // 'value'

// Check if key exists
console.log(weakMap.has(key)); // true

// Delete key-value pair
weakMap.delete(key);

// Check if key exists
console.log(weakMap.has(key)); // false
```

Summary

- **WeakSets**: Use when you need a collection of objects that should be garbage collected when no longer in use.
- **WeakMaps**: Use when you need to associate metadata or additional information with objects, and you want to ensure that this metadata does not prevent garbage collection of the objects.

These structures are particularly useful in scenarios where you need to manage memory efficiently and ensure that objects can be cleaned up when they are no longer needed. This lecture provides an introduction to working with strings in JavaScript, covering various methods and techniques. Here's a summary of the key points:

1. Basic String Operations

- **Accessing Characters**: Use bracket notation to access individual characters in a string.

```
const plane = "A320";
console.log(plane[0]); // "A"
console.log(plane[1]); // "3"
console.log(plane[2]); // "2"
```

- **Length Property**: Get the length of a string using the `.length` property.

```
const airline = "TAP Air Portugal";
console.log(airline.length); // 15
```

2. String Methods

- `indexOf()`: Finds the index of the first occurrence of a substring.

```
const airline = "TAP Air Portugal";
console.log(airline.indexOf("r")); // 6
console.log(airline.indexOf("Portugal")); // 8
console.log(airline.indexOf("portugal")); // -1 (case-sensitive)
```

- `lastIndexOf()`: Finds the index of the last occurrence of a substring.

```
console.log(airline.lastIndexOf("r")); // 10
```

- `slice()`: Extracts a substring from a string based on start and end indices.

```
console.log(airline.slice(4)); // "Air Portugal"
console.log(airline.slice(4, 7)); // "Air"
console.log(airline.slice(-7)); // "Portugal"
console.log(airline.slice(4, -1)); // "Air Portuga"
```

3. Practical Example: Checking Middle Seat

The example demonstrates a function that checks if an airplane seat is a middle seat (B or E) based on its seat code.

```
function checkMiddleSeat(seat) {
  const lastChar = seat.slice(-1);
  if (lastChar === "B" || lastChar === "E") {
    console.log("You got the middle seat 🛩");
  } else {
    console.log("You got lucky 🐞");
  }
}

checkMiddleSeat("23B"); // "You got the middle seat 🛩"
checkMiddleSeat("3E"); // "You got the middle seat 🛩"
checkMiddleSeat("12A"); // "You got lucky 🐞"
```

4. Behind the Scenes: String Methods

- **String Object Conversion**: JavaScript automatically converts strings to String objects when methods are called. This process is known as "boxing".

```
const str = "example";
console.log(typeof str); // "string"
console.log(str.toUpperCase()); // "EXAMPLE"
console.log(typeof str.toUpperCase()); // "string"
```

5. Summary

- Strings in JavaScript have methods similar to arrays for accessing and manipulating their content.
- Key methods include `indexOf()`, `lastIndexOf()`, and `slice()`, which help in finding positions and extracting substrings.
- Practical application of string methods can be seen in functions that perform specific tasks based on string content.

This foundational knowledge about strings will be applied throughout the course, and understanding these methods will greatly enhance your ability to work with text data in JavaScript.

String Methods: Changing Case and Practical Examples

In this lecture, we explored some fundamental string methods in JavaScript, particularly focusing on changing the case of strings and applying these methods in practical scenarios. Here's a summary of the key concepts covered:

1. Changing the Case of a String

- `toLowerCase()`: Converts the entire string to lowercase.
- `toUpperCase()`: Converts the entire string to uppercase.
- Example:

```
let airline = "TAP Air Portugal";
console.log(airline.toLowerCase()); // "tap air portugal"
console.log(airline.toUpperCase()); // "TAP AIR PORTUGAL"
```

2. Correcting Capitalization in Names

- Problem: A passenger's name is entered with incorrect capitalization.
- Solution: Convert the entire name to lowercase and then capitalize only the first letter.
- Example:

```
let passenger = "jOhN";
let passengerLower = passenger.toLowerCase(); // "john"
let passengerCorrect =
  passengerLower[0].toUpperCase() + passengerLower.slice(1); // "John"
```

- **Function Suggestion**: Create a function to fix capitalization for any given name.

3. Comparing Emails

- Problem: A user enters an email with extra spaces and incorrect capitalization.
- Solution: Normalize the email by converting it to lowercase and trimming whitespace.
- Example:

```
let loginEmail = " Hello@Jonas.IO \n";
let normalizedEmail = loginEmail.toLowerCase().trim(); // "hello@jonas.io"
```

- **Chaining Methods**: We can chain methods like `toLowerCase()` and `trim()` for efficiency.

4. Replacing Parts of Strings

- `replace()`: Replaces a specified value with another in a string.
- **Chaining** `replace()`: Multiple replacements can be done by chaining the `replace()` method.
- Example:

```
let priceGB = "288,97£";
let priceUS = priceGB.replace("£", "$").replace(",", "."); // "288.97$"
```

- **Replacing Words**: Replacing entire words, not just characters.

```
let announcement = "All passengers come to boarding door 23!";
let newAnnouncement = announcement.replace("door", "gate"); // "All
passengers come to boarding gate 23!"
```

- **Regular Expressions for Global Replacement**:
  - **Regular Expression with** `g` **flag**: Replaces all occurrences of a substring.
  - Example:

```
let newAnnouncement = announcement.replace(/door/g, "gate");
```

5. Boolean Methods: `includes()`, `startsWith()`, `endsWith()`

- `includes(substring)`: Checks if the string contains the specified substring.
- `startsWith(substring)`: Checks if the string starts with the specified substring.
- `endsWith(substring)`: Checks if the string ends with the specified substring.
- Example:

```

```

```
let plane = "Airbus A320neo";
console.log(plane.includes("A320")); // true
console.log(plane.startsWith("Airb")); // true
console.log(plane.endsWith("neo")); // true
```

6. Practical Example: Baggage Check

- Problem: Checking if a passenger's baggage contains prohibited items like a knife or a gun.
- Solution: Convert the baggage description to lowercase and check for prohibited items using `includes()`.
- Example:

```
function checkBaggage(items) {
  let baggage = items.toLowerCase();
  if (baggage.includes("knife") || baggage.includes("gun")) {
    console.log("You are not allowed on board");
  } else {
    console.log("Welcome aboard");
  }
}
checkBaggage("I have a laptop, some food and a pocket KNIFE"); // You are
not allowed on board
```

Key Takeaways:

- **String Methods**: `toLowerCase()`, `toUpperCase()`, `replace()`, `includes()`, `startsWith()`, `endsWith()` are essential tools for manipulating strings.
- **Practical Applications**: These methods are highly useful in real-world scenarios like normalizing user input and checking for specific content in strings.
- **Chaining Methods**: Combining multiple methods can make the code more concise and efficient.
- **Regular Expressions**: Useful for more complex string replacements, especially when dealing with multiple occurrences.

These methods are foundational in handling string data effectively in JavaScript, enabling you to perform various transformations and checks in your applications.

Working with Strings in JavaScript: Part 3

In this section, we'll delve into some of the most powerful string methods in JavaScript, including `split`, `join`, `padStart`, `padEnd`, and `repeat`. These methods are crucial for manipulating strings effectively.

1. The `split` Method

- The `split` method is used to divide a string into an array of substrings based on a specified separator.
- Example:

```
let str = "A+very+nice+string";
let parts = str.split("+");
console.log(parts); // ["A", "very", "nice", "string"]
```

- You can also use `split` to divide names:

```
let fullName = "John Doe";
let nameParts = fullName.split(" ");
console.log(nameParts); // ["John", "Doe"]
```

- **Destructuring with `split`:**

```
let [firstName, lastName] = fullName.split(" ");
console.log(firstName); // "John"
console.log(lastName); // "Doe"
```

2. The `join` Method

- The `join` method is the opposite of `split`, converting an array of strings into a single string, with elements separated by a specified string.
- Example:

```
let nameArray = ["Mr.", "John", "Doe"];
let newName = nameArray.join(" ");
console.log(newName); // "Mr. John Doe"
```

- Combining `split` and `join`:
  - These methods are powerful when used together, such as in formatting or capitalizing names.

3. Capitalizing Names

- Let's create a function to capitalize each word in a name:

```
function capitalizeName(name) {
  let names = name.split(" ");
  let namesUpper = [];
  for (let n of names) {
    namesUpper.push(n[0].toUpperCase() + n.slice(1));
  }
  return namesUpper.join(" ");
}
```

```
console.log(capitalizeName("jessica ann smith davis")); // "Jessica Ann
Smith Davis"
```

- **Alternative Approach:**
  - You can also use `replace`:

```
function capitalizeName(name) {
  return name
    .split(" ")
    .map((n) => n.replace(n[0], n[0].toUpperCase()))
    .join(" ");
}
```

4. The `padStart` and `padEnd` Methods

- These methods add padding to the start or end of a string until it reaches a desired length.
- Example:

```
let message = "Go to gate 23";
console.log(message.padStart(25, "+")); // "++++++++++Go to gate 23"
console.log(message.padEnd(30, "+")); // "Go to gate 23++++++++++++"
```

- **Real-World Application: Masking Credit Card Numbers**
  - This is a practical use of `padStart` to mask all but the last four digits of a credit card number:

```
function maskCreditCard(number) {
  let str = number + "";
  let last = str.slice(-4);
  return last.padStart(str.length, "*");
}
console.log(maskCreditCard(1234567812345678)); // "************5678"
```

5. The `repeat` Method

- The `repeat` method repeats a string a specified number of times.
- Example:

```
let weatherMessage = "Bad weather, all departures delayed.";
console.log(weatherMessage.repeat(3));
```

- **Simulating Planes in Line:**

```
function planesInLine(n) {
  console.log(`There are ${n} planes in line ${"✈".repeat(n)}`);
}
planesInLine(5); // "There are 5 planes in line ✈✈✈✈✈"
```

Summary

- **split and join**: Essential for breaking down and reconstructing strings.
- **padStart and padEnd**: Useful for formatting strings to a fixed length, such as masking sensitive information.
- **repeat**: Helps in creating repetitive string patterns efficiently.

String Manipulation Exercise: Formatting Flight Information

In this exercise, we'll take a string containing flight information and transform it into a readable format. The string represents multiple flights with details such as departure and arrival times, airport codes, and delay statuses. Our goal is to format this data into a legible output, displaying the flight information in a user-friendly way.

Step 1: Split the String into Individual Flights

The string contains multiple flights separated by a `+` sign. We'll use the `split` method to break the string into an array of individual flight information.

```
const flights =

"_Delayed_Departure;fao93766109;txl2133758440;11:25+_Arrival;bru0943384722;fao9376
6109;11:45+_Delayed_Arrival;hel7439299980;fao93766109;12:05+_Departure;fao93766109
;lis2323639855;12:30";

// Split the string into individual flights
const flightArray = flights.split("+");
console.log(flightArray);
```

Step 2: Loop Through Each Flight

We loop through the `flightArray` using a `for...of` loop, logging each flight to the console to ensure we have correctly split the string.

```
for (const flight of flightArray) {
  console.log(flight);
}
```

Step 3: Extract Information from Each Flight

Each flight's details are separated by a semicolon `;`. We'll split each flight string into its components: the type of flight (e.g., departure or arrival), the origin airport code, the destination airport code, and the time.

```
for (const flight of flightArray) {
  const [type, from, to, time] = flight.split(";");
  console.log(type, from, to, time);
}
```

Step 4: Format the Time

We'll format the time by replacing the colon `:` with the letter `H`.

```
const formattedTime = time.replace(":", "H");
```

Step 5: Replace Underscores in the Type

We replace any underscores `_` in the flight type with spaces to make the text more readable.

```
const formattedType = type.replaceAll("_", " ");
```

Step 6: Add a Delay Indicator

We'll add a red dot emoji 🔴 to indicate delayed flights. We use the `startsWith` method to check if the flight type begins with the word "Delayed."

```
const delayIndicator = formattedType.startsWith("Delayed") ? "🔴" : "";
```

Step 7: Format Airport Codes

The airport codes in the `from` and `to` fields include unnecessary numbers. We'll slice the first three characters of each code and convert them to uppercase.

```
const getCode = (str) => str.slice(0, 3).toUpperCase();
const formattedFrom = getCode(from);
const formattedTo = getCode(to);
```

Step 8: Assemble the Final String

We'll put everything together into a final output string and align it using the `padStart` method to make the output more visually appealing.

```
const output =
  `${delayIndicator} ${formattedType} from ${formattedFrom} to ${formattedTo}
(${formattedTime})`.padStart(
    36
  );
console.log(output);
```

Full Solution

Putting it all together, here's the complete solution:

```
const flights =

"_Delayed_Departure;fao93766109;txl2133758440;11:25+_Arrival;bru0943384722;fao9376
6109;11:45+_Delayed_Arrival;hel7439299980;fao93766109;12:05+_Departure;fao93766109
;lis2323639855;12:30";

const flightArray = flights.split("+");

for (const flight of flightArray) {
  const [type, from, to, time] = flight.split(";");

  const formattedTime = time.replace(":", "H");
  const formattedType = type.replaceAll("_", " ");
  const delayIndicator = formattedType.startsWith("Delayed") ? "🟤" : "";
  const getCode = (str) => str.slice(0, 3).toUpperCase();
  const formattedFrom = getCode(from);
  const formattedTo = getCode(to);

  const output =
    `${delayIndicator} ${formattedType} from ${formattedFrom} to ${formattedTo}
(${formattedTime})`.padStart(
      36
    );
  console.log(output);
}
```

Output

The final output will display the flight information in a well-formatted way:

```
🟤 Delayed Departure from FAO to TXL (11H25)
               Arrival from BRU to FAO (11H45)
```

```
🛑 Delayed Arrival from HEL to FAO (12H05)
            Departure from FAO to LIS (12H30)
```

Section 10 : Closure look at functions

Default Parameters in JavaScript (ES6)

In this section, we'll explore **default parameters** in JavaScript, a feature introduced in ES6 that simplifies how we define default values for function parameters.

1. Introduction to Default Parameters

Default parameters allow us to set default values for function parameters. This feature is particularly useful when we want to make some parameters optional, meaning that if no argument is provided for that parameter, the function will automatically use the default value.

2. Creating a Basic Function

Let's create a simple function `createBooking` that simulates making an airline booking. This function will take three parameters:

- `flightNum`: the flight number.
- `numPassengers`: the number of passengers (default is 1).
- `price`: the price of the booking (default is 199).

Initially, we'll write the function without using default parameters.

```
"use strict";

// Bookings array to store the bookings
const bookings = [];

// Function to create a booking
const createBooking = function (flightNum, numPassengers, price) {
  const booking = {
    flightNum,
    numPassengers,
    price,
  };
  console.log(booking);
  bookings.push(booking);
};

// Calling the function without providing all arguments
createBooking("LH123");
```

In this example, if you call `createBooking("LH123")`, both `numPassengers` and `price` will be `undefined`.

3. Implementing Default Parameters (ES5 vs ES6)

ES5 Method: Using Short-Circuiting

Before ES6, default parameters were usually implemented using short-circuiting with the `||` operator:

```
const createBooking = function (flightNum, numPassengers, price) {
  numPassengers = numPassengers || 1;
  price = price || 199;

  const booking = {
    flightNum,
    numPassengers,
    price,
  };
  console.log(booking);
  bookings.push(booking);
};
```

Here, if `numPassengers` or `price` is `undefined` (or any other falsy value like `0` or `""`), it will default to `1` or `199`, respectively.

ES6 Method: Simplified Default Parameters

With ES6, we can define default values directly in the function signature, making the code cleaner and more readable:

```
const createBooking = function (flightNum, numPassengers = 1, price = 199) {
  const booking = {
    flightNum,
    numPassengers,
    price,
  };
  console.log(booking);
  bookings.push(booking);
};

// Calling the function
createBooking("LH123"); // Defaults applied
createBooking("LH123", 2, 800); // Specified values used
```

In this version, the function defaults to `1` passenger and a price of `199` if no other values are provided.

4. Advanced Default Parameters

One powerful feature of ES6 default parameters is that they can be expressions, even using other parameter values.

```
const createBooking = function (
  flightNum,
  numPassengers = 1,
  price = numPassengers * 199
) {
  const booking = {
    flightNum,
    numPassengers,
    price,
  };
  console.log(booking);
  bookings.push(booking);
};

// Examples
createBooking("LH123", 2); // price = 2 * 199
createBooking("LH123", undefined, 1000); // Skipping numPassengers, custom price
```

In this example, the `price` is calculated dynamically based on the number of passengers. This flexibility allows for more complex and useful default values.

5. Skipping Default Parameters

To skip a parameter but still use the default value for it, you can pass `undefined`:

```
createBooking("LH123", undefined, 1000); // numPassengers defaults to 1
```

Passing `undefined` to a parameter is equivalent to not passing it at all, so the default value will be used.

6. Key Takeaways

- **Default Parameters**: Simplify function definitions by allowing parameters to have default values.
- **Flexibility**: Default values can be simple values or complex expressions based on other parameters.
- **Skipping Parameters**: Use `undefined` to skip a parameter and keep its default value.

```
# Passing Arguments into Functions in JavaScript

## Introduction

In this lecture, we'll delve into how arguments are passed into functions in
JavaScript, focusing on the differences between passing primitive types and
reference types (objects).

## Example: Flight Number and Passenger Object

Let's begin with a simple example. We'll define a flight number and a passenger
object:
```

```javascript
let flight = "LH234";
const jonas = {
  name: "Jonas Schmedtmann",
  passport: 123456789,
};
```

Next, we'll create a function `checkIn` that simulates a passenger checking in for a flight. The function will accept a `flightNum` and a `passenger` object:

```javascript
const checkIn = function (flightNum, passenger) {
  flightNum = "LH999"; // Simulating a change in flight number
  passenger.name = "Mr. " + passenger.name; // Adding 'Mr.' to the passenger's name

  if (passenger.passport === 123456789) {
    alert("Check in");
  } else {
    alert("Wrong passport");
  }
};
```

We'll then call the `checkIn` function:

```javascript
checkIn(flight, jonas);
console.log(flight); // LH234
console.log(jonas); // { name: 'Mr. Jonas Schmedtmann', passport: 123456789 }
```

Analysis

Primitive Types

- The `flight` variable is a **primitive type** (string).
- When `flight` is passed to the `checkIn` function, its value is **copied** into the `flightNum` parameter.
- Inside the function, changing `flightNum` does not affect the original `flight` variable, which remains `'LH234'`.

This behavior is because primitives are passed by **value**.

Reference Types (Objects)

- The `jonas` variable is an **object** (reference type).
- When `jonas` is passed to the `checkIn` function, a **reference** to the object in memory is passed, not the actual object.

- Inside the function, modifying the `passenger` object (which is just another reference to `jonas`) directly affects the original `jonas` object.

This occurs because objects are passed by **reference** to the memory address where the object is stored. However, it's important to note that JavaScript does not have true **pass by reference** like some other languages (e.g., C++). Instead, it passes the reference (which is a value) by value.

Demonstrating Potential Issues

Consider another function, `newPassport`, which changes the passport number of a person:

```
const newPassport = function (person) {
  person.passport = Math.trunc(Math.random() * 1000000000);
};

newPassport(jonas);
checkIn(flight, jonas); // Might now alert 'Wrong passport' due to the changed
passport number
```

In this scenario:

- The `newPassport` function modifies the `jonas` object.
- This change persists because the `jonas` reference is pointing to the same memory address.

Summary

- **Primitive Types**: Passed by value, so changes inside functions do not affect the original variable.
- **Reference Types (Objects)**: Passed by reference (but the reference itself is passed by value), meaning changes inside functions affect the original object.

Key Takeaways

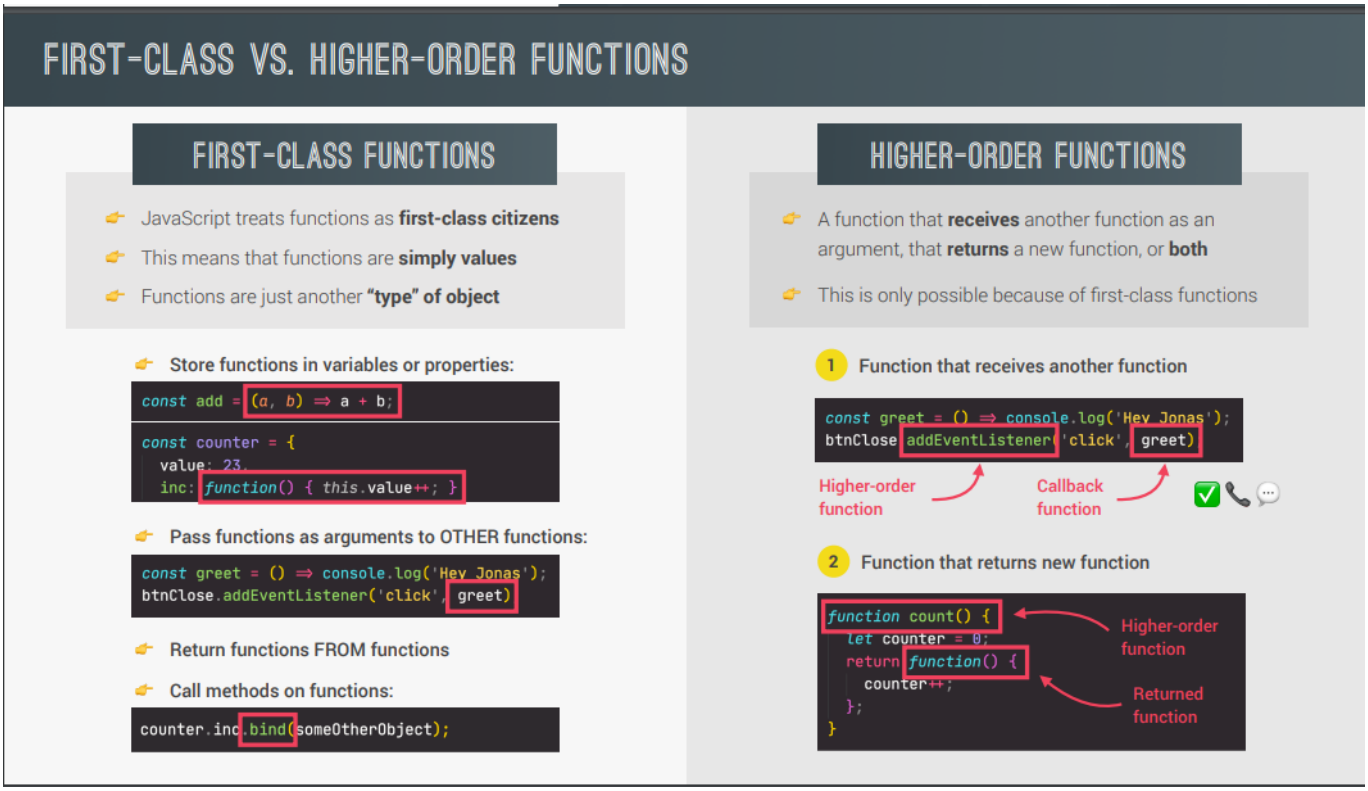
1. **Primitive vs. Reference Types**: Always remember the distinction between these two when passing variables into functions.
2. **Pass by Value vs. Pass by Reference**: JavaScript only has pass by value, even though it might seem like pass by reference for objects.
3. **Mutability**: Be cautious of mutating objects inside functions, as these changes will reflect outside the function.

First-Class Functions and Higher-Order Functions in JavaScript

Introduction

JavaScript is a language with first-class functions, a fundamental concept that opens up powerful programming paradigms. Understanding this concept is key to mastering JavaScript and writing more flexible,

reusable code.



What Are First-Class Functions?

In JavaScript, functions are treated as first-class citizens. This means that functions are treated like any other value in the language. They can be:

- **Stored in variables**: Just like numbers or strings.
- **Passed as arguments to other functions**: This allows for powerful abstractions like callback functions.
- **Returned from other functions**: This enables techniques like function factories or currying.
- **Stored in object properties**: Functions can be methods within objects.

The reason JavaScript can treat functions this way is that functions are actually objects in JavaScript. Since objects are values, functions are values too.

Examples of First-Class Functions

- **Storing in Variables**:

```
const greet = function () {
  console.log("Hello");
};
```

Here, the function is stored in a variable called `greet`.

- **Passing as Arguments**:

```
button.addEventListener("click", greet);
```

In this example, the `greet` function is passed as an argument to `addEventListener`.

- **Returning from Functions**:

```
function multiplyBy(factor) {
  return function (number) {
    return number * factor;
  };
}
```

This function returns another function.

Higher-Order Functions

Higher-order functions are functions that either:

1. **Receive a function as an argument**:
   - Example: `addEventListener` is a higher-order function because it takes a callback function as an argument.

```
button.addEventListener("click", greet);
```

Here, `addEventListener` is the higher-order function, and `greet` is the callback function.

2. **Return a new function**:
   - Example: The `multiplyBy` function returns a new function.

```
const double = multiplyBy(2);
console.log(double(5)); // 10
```

Difference Between First-Class Functions and Higher-Order Functions

- **First-Class Functions**: This is a language feature that indicates functions are values and can be manipulated like any other value (e.g., stored in variables, passed around).
- **Higher-Order Functions**: These are functions that take other functions as arguments or return them. They rely on the first-class function feature but represent a specific use case.

Conclusion

First-class functions are a foundational concept, while higher-order functions build on this idea to create more dynamic and reusable code.

Higher Order Functions in JavaScript: A Practical Example

In this lecture, we explored the concept of higher-order functions by creating our own function that accepts other functions as arguments. Here's a breakdown of the key points:

1. Creating Generic String Transformation Functions

- `oneWord` **Function:**
  - **Purpose:** Removes all spaces from a string and converts it to lowercase.
  - **Code:**

```
const oneWord = function (str) {
  return str.replace(/\s+/g, "").toLowerCase();
};
```

  - **Explanation:** The function uses `String.prototype.replace` with a regular expression to remove all spaces (`\s+`) globally (`g` flag) and converts the string to lowercase using `toLowerCase()`.

- `upperFirstWord` **Function:**
  - **Purpose:** Converts the first word of a string to uppercase.
  - **Code:**

```
const upperFirstWord = function (str) {
  const [first, ...others] = str.split(" ");
  return [first.toUpperCase(), ...others].join(" ");
};
```

  - **Explanation:** The function splits the string into an array of words, converts the first word to uppercase, and then joins the array back into a single string.

2. Creating a Higher Order Function

- `transformer` **Function:**
  - **Purpose:** Accepts a string and a transformation function, applies the function to the string, and logs the original and transformed strings.
  - **Code:**

```
const transformer = function (str, fn) {
  console.log(`Original string: ${str}`);
  console.log(`Transformed string: ${fn(str)}`);
  console.log(`Transformed by: ${fn.name}`);
};
```

  - **Explanation:** This function logs the original string, applies the transformation function (`fn`) to the string, logs the transformed string, and logs the name of the transformation function using

`fn.name`.

3. Demonstrating the Higher Order Function

- **Using `upperFirstWord`:**

```
transformer("JavaScript is the best", upperFirstWord);
```

Output:

```
Original string: JavaScript is the best
Transformed string: JAVASCRIPT is the best
Transformed by: upperFirstWord
```

- **Using `oneWord`:**

```
transformer("JavaScript is the best", oneWord);
```

Output:

```
Original string: JavaScript is the best
Transformed string: javascriptisthebest
Transformed by: oneWord
```

4. Callback Functions in JavaScript

- **Callback Functions:** Functions passed as arguments to other functions and executed later.
- **Real-World Examples:**
  - `addEventListener`:

```
const highFive = () => console.log("👋");
document.body.addEventListener("click", highFive);
```

  - `forEach`:

```
const names = ["Jonas", "Martha", "Adam"];
names.forEach(highFive);
```

5. Advantages of Callback Functions

- **Reusability:** Breaks code into reusable, interconnected parts.
- **Abstraction:** Hides the details of code implementation, allowing us to think at a higher level. For example, the `transformer` function abstracts away the details of how the string is transformed, leaving that responsibility to the passed-in functions (`oneWord`, `upperFirstWord`).

6. **Conclusion and Exercise**

- **Conclusion:** Callback functions are a vital part of JavaScript, allowing us to create flexible, reusable, and abstracted code.

Creating Functions that Return Functions in JavaScript

In this lecture, we explored the concept of creating a function that returns another function. This is a common pattern in JavaScript, especially in the context of functional programming. Let's break down the key points.

1. **Basic `greet` Function**

- **Purpose:** Create a function that takes a greeting message and returns a new function that accepts a name. The returned function then logs a greeting message to the console.
- **Code:**

```
function greet(greeting) {
  return function (name) {
    console.log(`${greeting} ${name}`);
  };
}
```

- **Explanation:**
  - The `greet` function takes a `greeting` as an argument and returns a new function.
  - The returned function takes `name` as an argument and logs the greeting message followed by the name.

2. **Using the `greet` Function**

- **Storing the Returned Function:**

```
const greeterHey = greet("Hey");
```

  - **Explanation:** The `greet('Hey')` call returns a function that expects a name. We store this returned function in the variable `greeterHey`.

- **Calling the Stored Function:**

```
greeterHey("Jonas");
greeterHey("Steven");
```

- **Output:**

```
Hey Jonas
Hey Steven
```

- **Explanation:** When `greeterHey` is called with a name, it logs the message using the greeting stored in the original `greet` function call.

3. Understanding Closures

- **Closures:** A closure is a feature in JavaScript where an inner function has access to variables in its outer function's scope, even after the outer function has returned. This is why `greeterHey` still remembers the `greeting` passed into `greet` even though `greet` has already finished execution.

4. Calling the Function Immediately

- **In-Line Function Calls:**

```
greet("Hello")("Jonas");
```

- **Output:**

```
Hello Jonas
```

- **Explanation:** This example demonstrates calling the returned function immediately after it's returned. The code `greet('Hello')` returns a function, and `('Jonas')` immediately calls that function.

5. Rewriting with Arrow Functions

- **Using Arrow Functions:**

```
const greetR = (greeting) => (name) => console.log(`${greeting} ${name}`);
```

- **Explanation:**
  - The outer arrow function takes `greeting` as an argument and returns an inner arrow function.
  - The inner arrow function takes `name` as an argument and logs the greeting message.
  - This syntax is more concise but can be harder to read, especially for those unfamiliar with arrow functions.

- **Testing the Arrow Function:**

```
greetR("Hi")("Jonas");
```

- **Output:**

```
Hi Jonas
```

6. **Why Use Functions that Return Functions?**

- **Abstraction:** Functions that return other functions allow for greater abstraction and reusability in code.
- **Functional Programming:** This pattern is essential in functional programming, enabling more modular and flexible code.
- **Practical Use:** While this example might seem simple, this technique is powerful in real-world applications, such as event handling, currying, and partial application.

Setting the `this` Keyword Manually in JavaScript

In this lecture, we revisit the `this` keyword and explore how to manually set it using different methods. Understanding how and why we might want to control the `this` keyword is essential when dealing with different contexts, especially in scenarios involving multiple objects that share common methods.

Creating an Airline Object

Let's start by creating a simple object for an airline, **Lufthansa**, with a method for booking flights. This object will include:

- The airline name (`airline`).
- The IATA code (`iataCode`).
- An array to store bookings (`bookings`).
- A `book` method that logs a booking to the console and adds it to the `bookings` array.

```
const lufthansa = {
  airline: "Lufthansa",
  iataCode: "LH",
  bookings: [],
  book(flightNumber, passengerName) {
    console.log(
      `${passengerName} booked a seat on ${this.airline} flight
${this.iataCode}${flightNumber}`
    );
    this.bookings.push({
      flight: `${this.iataCode}${flightNumber}`,
      name: passengerName,
    });
  },
};
```

Using the `book` Method

Now, we can use the `book` method to book flights for different passengers:

```
lufthansa.book(239, "Uday Kumar Kathari");
lufthansa.book(635, "John Smith");
```

This works as expected, logging the bookings and storing them in the `bookings` array.

Creating Another Airline Object

Let's create another airline, **Eurowings**, with a similar structure:

```
const eurowings = {
  airline: "Eurowings",
  iataCode: "EW",
  bookings: [],
};
```

Avoiding Code Duplication

Instead of copying the `book` method from **Lufthansa** to **Eurowings**, we can reuse the method by storing it in an external function:

```
const book = lufthansa.book;
```

However, if we try to book a flight using this `book` function:

```
book(23, "Sarah Williams");
```

We encounter an error: `Cannot read property 'airline' of undefined`. This happens because the `this` keyword inside the `book` function now points to `undefined`, as it's a regular function call, not a method call on an object.

Manually Setting `this` with `call`

To fix this, we can use the `call` method, which allows us to explicitly set the `this` keyword:

```
book.call(eurowings, 23, "Sarah Williams");
```

This sets `this` to `eurowings`, and the booking is successfully added to **Eurowings**' `bookings` array.

Using `call` with Other Objects

We can use `call` with any object that shares the same properties. For instance, let's book a flight on **Lufthansa**:

```
book.call(lufthansa, 239, "Mary Cooper");
```

This method is versatile and allows us to reuse functions across different objects by manually setting the `this` context.

Using `apply` Method

The `apply` method is similar to `call`, but it takes an array of arguments instead of a list:

```
const flightData = [583, "George Cooper"];
book.apply(eurowings, flightData);
```

This achieves the same result as `call`, but it's less commonly used in modern JavaScript due to the spread operator.

Replacing `apply` with the Spread Operator

Instead of using `apply`, we can achieve the same result using the spread operator with `call`:

```
book.call(eurowings, ...flightData);
```

This approach is more readable and aligns with modern JavaScript practices.

Key Takeaways

- The `this` keyword can be manually set using the `call`, `apply`, and `bind` methods.
- `call` allows you to pass arguments individually, while `apply` accepts an array of arguments.
- The spread operator can replace `apply` in many cases, making the code more readable.
- Manually controlling `this` is useful when reusing methods across different objects, such as in the case of multiple airline objects.

Understanding the `bind` Method in JavaScript

The `bind` method in JavaScript is an essential tool that allows you to create a new function with a specific `this` value. This can be particularly useful when working with object methods that you want to pass around or use as callbacks, where the `this` context might otherwise be lost or altered.

Key Points about `bind`:

1. **Setting the `this` Context**:

- The primary purpose of `bind` is to set the `this` keyword to a specific object for any function. Unlike `call` and `apply`, which immediately invoke the function, `bind` returns a new function where the `this` value is permanently set to the object you specify.

2. **Does Not Invoke the Function Immediately**:
   - `bind` does not execute the function right away. Instead, it returns a new function where the `this` context and any arguments you provide are pre-set.
3. **Partial Application**:
   - `bind` allows you to set some or all of the function's arguments in advance, a concept known as partial application. This makes the function easier to use in repetitive contexts where certain arguments remain constant.

Example: Using `bind` in an Airline Booking System

Let's revisit the airline booking example to understand `bind` in action.

```
const lufthansa = {
  airline: "Lufthansa",
  iataCode: "LH",
  bookings: [],
  book(flightNum, name) {
    console.log(
      `${name} booked a seat on ${this.airline} flight
${this.iataCode}${flightNum}`
    );
    this.bookings.push({ flight: `${this.iataCode}${flightNum}`, name });
  },
};

const eurowings = {
  airline: "Eurowings",
  iataCode: "EW",
  bookings: [],
};

// Creating a new function with `bind`:
const bookEW = lufthansa.book.bind(eurowings);
bookEW(23, "Steven Williams");

// Output: Steven Williams booked a seat on Eurowings flight EW23
```

In this example:

- The `book` method is originally tied to the `lufthansa` object.
- We used `bind` to create `bookEW`, a new function where the `this` keyword is permanently set to `eurowings`.
- When `bookEW` is called, it behaves as if it was always a method of `eurowings`.

Advanced Usage: Partial Application with `bind`

You can also preset arguments using `bind`, which allows you to create specialized versions of functions.

```
const bookEW23 = lufthansa.book.bind(eurowings, 23);
bookEW23("Jonas Smith");
bookEW23("Martha Cooper");

// Output:
// Jonas Smith booked a seat on Eurowings flight EW23
// Martha Cooper booked a seat on Eurowings flight EW23
```

Here:

- `bookEW23` is a version of the `book` function where the flight number is always `23`.
- You only need to provide the passenger's name when calling `bookEW23`.

`bind` with Event Listeners

When using methods as event handlers, `bind` is crucial for maintaining the correct `this` context.

```
lufthansa.planes = 300;
lufthansa.buyPlane = function () {
  console.log(this);
  this.planes++;
  console.log(this.planes);
};

document
  .querySelector(".buy")
  .addEventListener("click", lufthansa.buyPlane.bind(lufthansa));
```

Without `bind`, `this` inside `buyPlane` would refer to the button element, not the `lufthansa` object. Using `bind`, we ensure that `this` always points to `lufthansa`, so the method works as intended.

Creating a Function with `bind` Instead of Returning Another Function

A common use case for `bind` is when you want to avoid writing a function that returns another function. For instance, consider a function to add tax:

```
function addTax(rate, value) {
  return value + value * rate;
}

const addVAT = addTax.bind(null, 0.23);

console.log(addVAT(100)); // 123
console.log(addVAT(200)); // 246
```

Here, `addVAT` is a new function where the tax rate is fixed at 23%. This approach is cleaner and avoids the need to write nested functions.

Challenge Solution: Function Returning Another Function

If you wanted to achieve a similar effect without using `bind`, you could write:

```
function addTaxRate(rate) {
  return function (value) {
    return value + value * rate;
  };
}

const addVAT2 = addTaxRate(0.23);
console.log(addVAT2(100)); // 123
console.log(addVAT2(200)); // 246
```

This approach works similarly but requires more code. Using `bind` simplifies things by allowing you to create specialized functions with less effort.

Conclusion

The `bind` method is a powerful tool for setting the `this` context and partially applying functions. Whether you're dealing with object methods, event listeners, or simply want to create a more specific version of a function, `bind` is an invaluable method in JavaScript. Understanding how to use it effectively will make your code more flexible and easier to manage. Here's a breakdown of the solution to the challenge:

Solution for Poll Application

1. Create a method called 'registerNewAnswer' on the 'poll' object. The method does 2 things: 1.1. Display a prompt window for the user to input the number of the selected option. The prompt should look like this: What is your favourite programming language? 0: JavaScript 1: Python 2: Rust 3: C++ (Write option number)

   1.2. Based on the input number, update the answers array. For example, if the option is 3, increase the value AT POSITION 3 of the array by 1. Make sure to check if the input is a number and if the number makes sense (e.g answer 52 wouldn't make sense, right?)

2. Call this method whenever the user clicks the "Answer poll" button.

3. Create a method 'displayResults' which displays the poll results. The method takes a string as an input (called 'type'), which can be either 'string' or 'array'. If type is 'array', simply display the results array as it is, using console.log(). This should be the default option. If type is 'string', display a string like "Poll results are 13, 2, 4, 1".

4. Run the 'displayResults' method at the end of each 'registerNewAnswer' method call.

HINT: Use many of the tools you learned about in this and the last section 😉

BONUS: Use the 'displayResults' method to display the 2 arrays in the test data. Use both the 'array' and the 'string' option. Do NOT put the arrays in the poll object! So what shoud the this keyword look like in this situation?

BONUS TEST DATA 1: [5, 2, 3] BONUS TEST DATA 2: [1, 5, 3, 9, 6, 1]

GOOD LUCK 😀 */

1. Define the `poll` Object

```
const poll = {
  question: "What is your favourite programming language?",
  options: ["JavaScript", "Python", "Rust", "C++"],
  answers: new Array(4).fill(0), // Initialize with zeros for each option
  registerNewAnswer() {
    // Display prompt
    const answer = Number(
      prompt(
        `${this.question}\n${this.options
          .map((opt, i) => `${i}: ${opt}`)
          .join("\n")}\n(Write option number)`
      )
    );

    // Validate and update answers
    if (!Number.isNaN(answer) && answer >= 0 && answer < this.answers.length) {
      this.answers[answer]++;
      this.displayResults(); // Call displayResults method
    }
  },
  displayResults(type = "array") {
    if (type === "string") {
      console.log(`Poll results are ${this.answers.join(", ")}`);
    } else {
      console.log(this.answers);
    }
  },
};
```

2. Attach Event Listener to the Button

Ensure that you have an HTML element with a class of `poll` for the button. Then add an event listener to this button:

```
document.querySelector(".poll").addEventListener("click", function () {
  poll.registerNewAnswer();
});
```

3. Bonus: Display Results with Custom Data

To use the `displayResults` method with custom arrays, you can use `call` to set the `this` context:

```
const BONUS_DATA1 = [5, 2, 3];
const BONUS_DATA2 = [1, 5, 3, 9, 6, 1];

poll.displayResults.call({ answers: BONUS_DATA1 }, "array");
poll.displayResults.call({ answers: BONUS_DATA2 }, "string");
```

Explanation

1. `registerNewAnswer` **Method**:
    - Prompts the user with a question and options.
    - Converts the user input to a number and validates it.
    - Updates the `answers` array if the input is valid.
    - Calls `displayResults` to show the updated poll results.
2. `displayResults` **Method**:
    - Displays results either as an array or a formatted string based on the `type` argument.
3. **Event Listener**:
    - Attaches a click event listener to a button with the class `poll` to trigger the `registerNewAnswer` method.
4. **Bonus**:
    - Uses `Function.prototype.call` to set the `this` context for the `displayResults` method to custom data arrays.

This setup allows the poll to dynamically handle user input and display results in different formats.

Immediately Invoked Function Expressions (IIFE)

Definition: An Immediately Invoked Function Expression (IIFE) is a design pattern in JavaScript where a function is defined and executed immediately. The function executes right after it is created and doesn't persist beyond its execution.

Purpose:

1. **Single Execution:** Useful for running a function once and only once.
2. **Scope Creation:** Helps in creating a new scope for variables, thereby avoiding polluting the global scope and protecting data from accidental overwrites.

Basic Syntax:

```
(function () {
  // Code here runs immediately
})();
```

Explanation:

1. **Function Expression:** By wrapping the function in parentheses, you convert the function declaration into an expression. This tells JavaScript to treat it as a function expression rather than a function declaration.
2. **Immediate Invocation:** After the function is defined as an expression, it is immediately invoked by appending `()`.

Example:

```
(function () {
  console.log("This function runs immediately.");
})();
```

Arrow Function Version:

```
(() => {
  console.log("This arrow function runs immediately.");
})();
```

Why Use IIFE?

1. **Scope Isolation:** Variables declared inside an IIFE are not accessible outside of it. This helps in keeping the global scope clean and avoids conflicts.

   ```
   (function () {
     var privateVar = "I am private";
     console.log(privateVar); // Accessible here
   })();
   console.log(privateVar); // Error: privateVar is not defined
   ```

2. **Data Encapsulation:** It allows data to be encapsulated and hidden from the global scope.

   ```
   (function () {
     var privateData = "Secret";
     window.publicData = "Visible";
   })();
   console.log(publicData); // "Visible"
   console.log(privateData); // Error: privateData is not defined
   ```

Modern Alternatives:

1. **Block Scope with `let` and `const`:** With ES6, `let` and `const` provide block scope, which can be used to achieve similar scope isolation.

```
{
  let blockScopedVar = "I'm block-scoped";
  console.log(blockScopedVar); // Accessible here
}
console.log(blockScopedVar); // Error: blockScopedVar is not defined
```

2. **Module Pattern:** ES6 modules provide a more robust solution for data encapsulation and scope management.

Conclusion: While IIFEs are less commonly used in modern JavaScript due to the advent of block scoping with `let` and `const`, and ES6 modules, they are still relevant for scenarios where immediate execution of a function is required. They serve as a good example of functional programming techniques and data encapsulation.

Closures in JavaScript

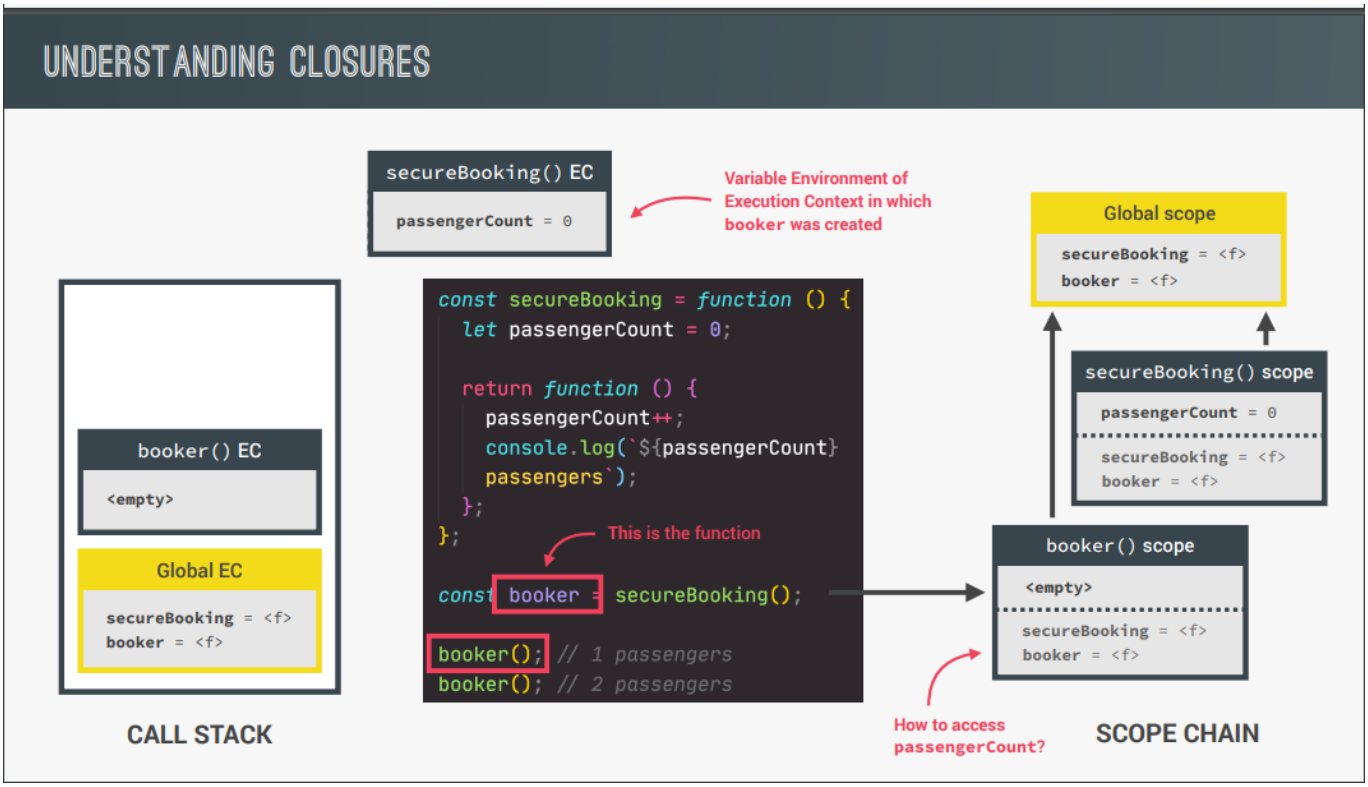
JavaScript functions have an almost mystical feature known as **closures**, which many developers struggle to fully understand. However, with a solid grasp of concepts like execution context, the call stack, and the scope chain, closures can be understood as a fascinating and integral part of the language.

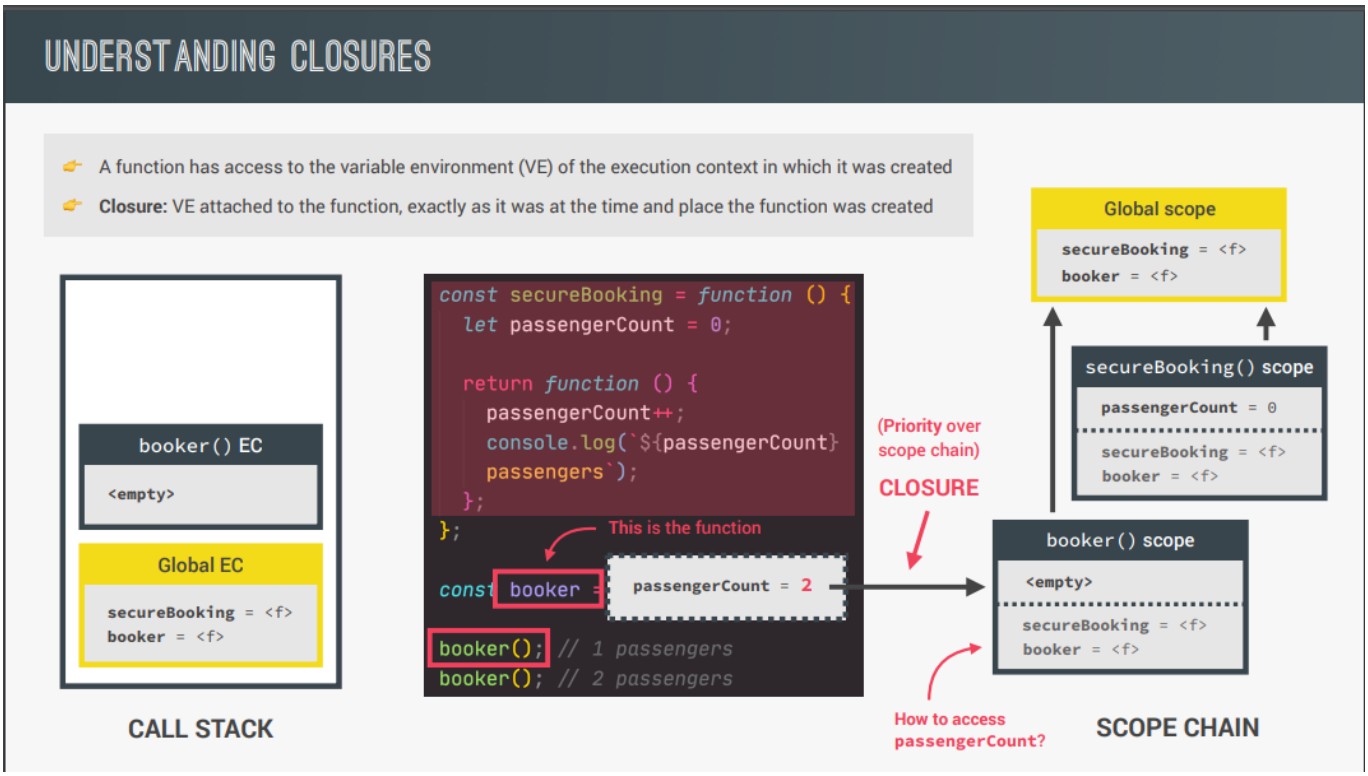
What Are Closures?

Closures are not something we create manually, like arrays or functions. Instead, they happen automatically in certain situations. The key is recognizing these situations and understanding how closures operate in those contexts.

Example: Secure Booking Function

Let's create a function called `secureBooking`, which will demonstrate a closure in action:






```
function secureBooking() {
  let passengerCount = 0;

  return function () {
    passengerCount++;
    console.log(`${passengerCount} passengers`);
  };
}
```

```
const booker = secureBooking();
```

Here, `secureBooking` returns a new function that updates the `passengerCount` variable defined in its parent function. This returned function is stored in the `booker` variable.

Analyzing the Execution Context

1. **Global Execution Context**: Before calling `secureBooking`, the code is running in the global execution context, which contains the `secureBooking` function.
2. **Calling `secureBooking`**: When `secureBooking` is executed, a new execution context is created, which includes its local variables like `passengerCount`.
3. **Returning the Inner Function**: The inner function, which increments `passengerCount`, is returned and stored in `booker`. The global context now contains this `booker` function.

Once `secureBooking` finishes execution, its execution context is removed from the call stack. However, the returned `booker` function still has access to `passengerCount`. This is where the closure comes into play.

The Closure in Action

Even after the `secureBooking` function has finished executing and its execution context is gone, the `booker` function still has access to `passengerCount`. This happens because:

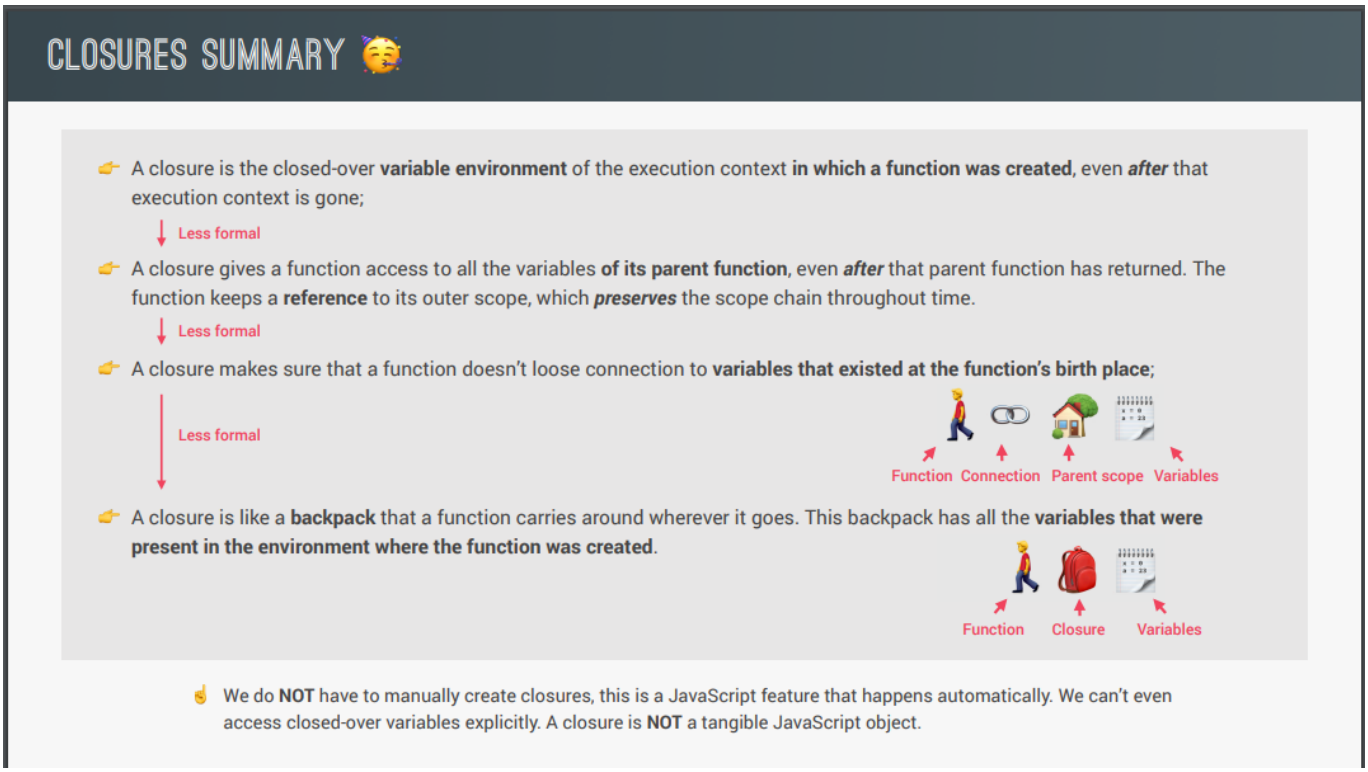
- **Closure**: A closure makes a function remember the variables that were present in its lexical environment when it was created, even after the function's parent scope is gone.

For example, calling `booker()` three times will output:

```
booker(); // 1 passengers
booker(); // 2 passengers
booker(); // 3 passengers
```

The `booker` function increments `passengerCount` with each call, despite `secureBooking` having completed its execution. This is possible because `booker` maintains a reference to the `passengerCount` variable through the closure.

Key Takeaways



What is a Closure?

A closure is a feature in JavaScript where an inner function has access to the variables and scope of an outer function, even after the outer function has finished executing. This allows the inner function to "remember" the environment in which it was created.

1. **Definition**: A closure is the closed-over variable environment of the execution context in which a function was created, even after that execution context is gone.
2. **Function Access**: A closure allows a function to access all variables from its parent function, even after the parent function has returned.
3. **Preserved Scope Chain**: The scope chain is preserved through the closure, ensuring that the function retains access to its birthplace variables.
4. **Backpack Analogy**: Think of a closure as a backpack that a function carries around, containing all the variables from its creation environment.

Visualizing Closures

You can observe closures in action using `console.dir` in the browser console:

```
console.dir(booker);
```

This will show the function's internal properties, including the closure, which contains the `passengerCount` variable.

Conclusion

Closures are a powerful feature in JavaScript, allowing functions to maintain a connection to their original scope even after that scope has been destroyed. Here are the two situations where closures appear, and the

explanation behind them:

Situation 1: Reassigning Functions

We don't always need to return a function to create a closure. Here's an example where closures are created simply by reassigning function values:

```
let f;

const g = function () {
  const a = 23;
  f = function () {
    console.log(a * 2);
  };
};

const h = function () {
  const b = 777;
  f = function () {
    console.log(b * 2);
  };
};

g(); // Assigns the first function to 'f'
f(); // Outputs 46 (23 * 2)

h(); // Reassigns a new function to 'f'
f(); // Outputs 1554 (777 * 2)
```

Explanation:

- **Closure in `g`:** When `g()` is called, the function `f` gets assigned, closing over the variable `a`. Even after `g` finishes, `f` retains access to `a` through closure.
- **Closure in `h`:** Later, when `h()` is called, `f` is reassigned to a new function that closes over the variable `b` instead. The closure now holds a reference to `b`, and `a` is no longer part of the closure.
- **Key takeaway:** This shows that closures can "change" when functions are reassigned. The new function closes over variables in its new lexical environment.

Situation 2: Timers (setTimeout)

In this situation, we create a closure inside a timer callback function.

```
function boardPassengers(n, wait) {
  const perGroup = n / 3;

  setTimeout(function () {
    console.log(`We are now boarding all ${n} passengers`);
```

```
    console.log(`There are 3 groups, each with ${perGroup} passengers`);
  }, wait * 1000);

  console.log(`Boarding will start in ${wait} seconds`);
}

boardPassengers(180, 3);
```

Explanation:

- **Closure with `setTimeout`:** The `setTimeout` function uses a callback that executes after a delay (3 seconds in this case). When the callback is finally executed, the outer function `boardPassengers` has already completed. However, the callback still has access to the `n` and `perGroup` variables from `boardPassengers` due to closure.
- **Priority of Closures over Scope Chain:** If there's a global `perGroup` variable, the callback will still prefer the local `perGroup` within `boardPassengers` because of closure. This proves that closures take priority over the normal scope chain.

Conclusion:

In both examples:

- The first example demonstrates that closures can form even when functions are reassigned.
- The second example shows how closures work in asynchronous code like timers.

Both scenarios illustrate that closures are created whenever a function needs access to variables from its birth environment, even if those variables are no longer in scope.

Section 14 : Object Oriented Programming with JavaScript.

Object-Oriented Programming (OOP) - A High-Level Overview

1. Introduction to Object-Oriented Programming (OOP)

- **Definition**: Object-Oriented Programming (OOP) is a programming paradigm centered around the concept of "objects". It is a structured approach to writing and organizing code.
- **Programming Paradigm**: A style or methodology for how code is written and organized. OOP emphasizes the use of objects, which encapsulate data and methods (behavior), to model real-world concepts or abstract components.

2. Why OOP?

- **Solves Spaghetti Code**: Before OOP, codebases often became disorganized (spaghetti code) with unrelated pieces of code scattered across functions and global scope. This made large codebases hard to maintain and extend.

- **Flexible & Maintainable**: OOP provides a way to structure code that makes it easier to maintain, extend, and scale large software projects.
- **Popular Paradigm**: OOP is one of the most widely used paradigms today, especially in large-scale software engineering, thanks to its ability to model complex applications using reusable objects.

3. Objects in OOP

- **Objects**: Represent real-world entities or abstract concepts. Objects contain:
  - **Properties (Data)**: Attributes of the object (e.g., a user object may have properties like `username`, `password`).
  - **Methods (Behavior)**: Functions that operate on the object's properties (e.g., a `login()` method).
- **Self-Contained Units**: Objects are self-contained blocks of code that combine related data and behavior. These are used as building blocks for applications.
- **Interaction via Public Interface (API)**: Objects interact with one another via methods exposed as part of their public API. This defines how external code interacts with an object.

4. Comparison with Functional Programming

- **OOP vs Functional Programming**: Both paradigms aim to avoid disorganized (spaghetti) code. However, they approach the problem differently:
  - **OOP**: Focuses on objects and stateful interactions.
  - **Functional Programming**: Focuses on functions and avoids shared state or mutable data.
- OOP is one paradigm, and functional programming is another, each with its advantages and use cases. Both can coexist in modern software development.

Classes and Instances

1. Classes in OOP

- **Blueprint for Objects**: A class is like a blueprint or template for creating objects. It defines what properties and methods an object will have.
  - Example: A `User` class might define properties like `username`, `email`, and methods like `login()`, `sendMessage()`.
- **Abstract Concept**: A class itself is not an object. It only describes how objects are created and what attributes and behaviors they will have.

2. Instances of Classes

- **Object Creation**: When we create an object from a class, we call it an **instance** of that class.
  - Example: We might create several user objects (instances of the `User` class), each with different `username` and `email` values.
- **Real-world analogy**: Think of a class as an architectural blueprint for a house. Each actual house built from that blueprint is an instance.

3. Example: Fictional `User` Class

```
User Class
- Properties: username, email, password
- Methods: login(), sendMessage()
```

- The class provides the structure for a user but does not contain real data. An instance, however, will have real data like a specific `username`, `email`, and functionality to log in or send messages.

Four Fundamental Principles of OOP

1. Abstraction

- **Definition**: Abstraction means focusing on the essential details while ignoring irrelevant complexities.
- **Purpose**: It allows developers to manage complexity by simplifying interactions and hiding unnecessary internal details from the user.
- **Example**:
  - A phone's user interface abstracts away all the complex operations (like controlling voltage or managing internal components). As users, we only interact with buttons and screens, while the internal processes are hidden.
  - In OOP, the object methods present a simplified interface to interact with, without exposing how they work internally.

2. Encapsulation

- **Definition**: Encapsulation is the practice of keeping an object's internal state (data) private and only exposing specific methods to interact with that state.
- **Purpose**: Encapsulation protects the object's internal data from being accessed or modified directly by external code, reducing the risk of unintended side effects or bugs.
- **Example**:
  - A `User` object might have a private `password` field that cannot be accessed directly. Instead, we use a `login()` method that validates the password internally.
  - Methods or properties marked as **private** are only accessible from within the object, while **public** methods form the object's API for outside interaction.

3. Inheritance

- **Definition**: Inheritance allows a new class to inherit properties and methods from an existing class.
- **Purpose**: This promotes code reusability and allows us to create hierarchical relationships between objects.
- **Example**:
  - A `User` class can be extended by an `Admin` class, where the admin inherits properties like `username` and `password` from `User`, but adds new methods like `banUser()`.

```
User Class
- username
```

```
- password
- login()

Admin Class (inherits from User)
- banUser()
```

The `Admin` class has all the functionality of the `User` class plus additional features specific to administrators.

4. Polymorphism

- **Definition**: Polymorphism allows objects of different classes to be treated as objects of a common parent class, enabling a single interface to represent different underlying forms (types).
- **Purpose**: It provides flexibility by allowing methods to be implemented in different ways depending on the object calling them.
- **Example**:
    - A parent class `Shape` might have a method `calculateArea()`. Subclasses like `Circle`, `Square`, or `Rectangle` implement this method differently based on their shape.

```
Shape Class
- calculateArea()

Circle Class (inherits from Shape)
- calculateArea() [calculation specific to a circle]

Square Class (inherits from Shape)
- calculateArea() [calculation specific to a square]
```

Even though `Circle` and `Square` are different objects, they can both be treated as a `Shape`, and their `calculateArea()` method can be called polymorphically.

Designing Classes

- **Modeling Real-world Data**: One of the key challenges in OOP is how to map real-world data and behaviors into classes. There isn't a single "correct" way to design a class, but the four fundamental principles of OOP (abstraction, encapsulation, inheritance, and polymorphism) guide developers toward good class design.
- **Example**: When modeling a `User`, consider what data and behavior are important for the object. For example:
    - **Data**: `username`, `password`, `email`
    - **Behavior**: `login()`, `sendMessage()`

Key Takeaways

1. **Object-Oriented Programming (OOP)** is a popular programming paradigm focused on organizing code into objects that combine data and behavior.
2. **Classes** are blueprints for creating objects. **Instances** are real objects created from these classes.
3. The four fundamental principles of OOP are:
    - **Abstraction**: Hiding complex details and focusing on what's essential.
    - **Encapsulation**: Keeping internal data private and exposing a public interface.
    - **Inheritance**: Reusing code by creating new classes from existing ones.
    - **Polymorphism**: Allowing objects to be treated as instances of a common superclass, with different behavior based on the object's specific class.
4. **OOP** helps organize code in a more maintainable and scalable way, especially in large-scale applications.

In-Depth Notes on Object-Oriented Programming in JavaScript

1. **Introduction to OOP in JavaScript**

- **Object-Oriented Programming (OOP)** in JavaScript differs from traditional OOP models like those seen in Java, C++, or Python.
- JavaScript has its own unique way of implementing OOP, which is based on **prototypes**, not classes (though modern JavaScript syntax introduces classes for convenience).
- **Prototypal Inheritance** is the key mechanism for object-oriented behavior in JavaScript.

2. **Review: Classical OOP Concepts**

- **Class**: A blueprint or template used to create objects (instances). It's a theoretical model, akin to a blueprint for a house.
- **Instance**: An object created from a class. Each instance is an actual object in the program that can be used to perform operations.
- **Instantiation**: The process of creating an instance from a class.

In traditional OOP, classes define the structure and behavior (methods and properties) of objects.

3. **How OOP Works in JavaScript**

- **Prototypes**: In JavaScript, objects are linked to **prototype objects**. Each object has a prototype that contains methods and properties.
- **Prototypal Inheritance**: Objects inherit methods and properties from their linked prototype object.
    - For example, if an object is linked to a prototype with a `map` method, the object can use that method even though the method is not defined directly on the object.
- **Delegation**: Objects delegate their behavior (methods and properties) to the prototype. This means that when a method is called on an object, JavaScript looks for it in the prototype if it's not directly defined on the object.

4. **Key Terminology**

- **Inheritance vs. Delegation**:
    - **Prototypal Inheritance** in JavaScript is different from classical inheritance. Instead of copying behavior from one class to another, objects **delegate** behavior to their prototype.

    - In classical OOP, methods are copied from a parent class to the child class, while in prototypal inheritance, the behavior is accessed through the prototype chain.
- **Behavior**: In OOP, behavior typically refers to methods. So, when we say an object delegates its behavior, we're referring to how it accesses methods via its prototype.

5. **Examples of Prototypal Inheritance**

- **Array Example**:
    - In JavaScript, when we use array methods like `.map()`, those methods are not defined on the array itself but on the `Array.prototype` object.
    - All arrays in JavaScript are linked to the `Array.prototype`, which contains methods like `.map()`, `.filter()`, and others. This is an example of prototypal inheritance in action.

6. **How Prototypes Work in Practice**

- Each time you create an object (like an array), it is linked to a prototype.
- Methods like `.map()` are accessible because the object delegates the behavior to its prototype.

7. **Implementing OOP in JavaScript**

There are **three main ways** to implement object-oriented programming in JavaScript:

a) Constructor Functions

- This is the traditional method for creating objects programmatically in JavaScript.
- A **constructor function** is used to create and initialize objects. It sets the prototype of the new object to allow inheritance of methods and properties.
- For example, built-in objects like `Array`, `Map`, and `Set` are implemented using constructor functions.

b) ES6 Classes

- Introduced in ES6, **classes** provide a modern syntax for creating objects. However, they are just **syntactic sugar** over constructor functions.
- **Behind the scenes**, ES6 classes still use constructor functions and prototypal inheritance.
- The class syntax is more readable and beginner-friendly but functions the same as constructor functions.

c) Object.create()

- This is a simpler way to create an object and directly link it to a prototype object.
- It's less commonly used compared to constructor functions and ES6 classes, but it's the most straightforward method for linking objects to prototypes.

8. **Four OOP Principles in JavaScript**

- The four key principles of object-oriented programming—**abstraction**, **encapsulation**, **inheritance**, and **polymorphism**—still apply to JavaScript, even with its prototypal inheritance model.
- As you implement OOP concepts using JavaScript's prototypes, these principles are still valid and important to understand.

9. **Summary**

- JavaScript has its own unique OOP model based on prototypes, known as **prototypal inheritance**.
- There are three main ways to implement OOP in JavaScript: **constructor functions**, **ES6 classes**, and **Object.create()**.
- **Prototypes** enable objects to inherit methods and properties, and this mechanism is key to understanding JavaScript's OOP structure.
- Although JavaScript's OOP is different from classical OOP, the core principles (abstraction, encapsulation, inheritance, polymorphism) remain relevant. To implement object-oriented programming (OOP) in JavaScript, we start with **constructor functions**. Constructor functions allow us to create objects programmatically, as opposed to using simple object literals. Let's walk through the steps to create and use constructor functions, covering key concepts like object creation, the `this` keyword, and object properties.

Key Concepts of Constructor Functions

A **constructor function** is a regular function with a special purpose: it's used to construct objects. The distinguishing feature is that we call it with the `new` keyword. This triggers a set of actions to create a new object and return it.

1. Creating a Constructor Function

We use constructor functions to build objects. By convention, constructor function names start with a capital letter.

```
function Person(firstName, birthYear) {
  this.firstName = firstName;
  this.birthYear = birthYear;
}
```

In this example, `Person` is a constructor function. It accepts `firstName` and `birthYear` as parameters and assigns these values to properties of `this`.

2. The `new` Operator

When we call a constructor function with `new`, four things happen behind the scenes:

1. A new empty object `{}` is created.
2. The constructor function is called, and the `this` keyword is set to the newly created object.
3. The newly created object is linked to the constructor function's prototype.
4. The new object is returned from the constructor function automatically.

```
const jonas = new Person("Jonas", 1991);
console.log(jonas); // { firstName: 'Jonas', birthYear: 1991 }
```

Here, `new Person('Jonas', 1991)` creates an object `jonas` with `firstName` and `birthYear` properties.

3. Creating Multiple Instances

You can create as many objects as you like using the constructor function, each with its own set of properties.

```
const matilda = new Person("Matilda", 2017);
const jack = new Person("Jack", 1975);

console.log(matilda); // { firstName: 'Matilda', birthYear: 2017 }
console.log(jack); // { firstName: 'Jack', birthYear: 1975 }
```

Each object (`jonas`, `matilda`, and `jack`) is an instance of the `Person` constructor function.

4. Checking Instances

To verify that an object is an instance of a constructor, you can use the `instanceof` operator:

```
console.log(jonas instanceof Person); // true
console.log(jack instanceof Person); // true
```

This returns `true` because `jonas` and `jack` were created using the `Person` constructor function.

Instance Properties

The properties we define within the constructor function, like `firstName` and `birthYear`, are called **instance properties** because they are unique to each instance of the object created from the constructor.

Adding Methods to Constructor Functions

We can also add methods to constructor functions. However, defining methods inside the constructor function is not a good practice because each instance would have its own copy of the method, which can hurt performance when many instances are created.

For example:

```
function Person(firstName, birthYear) {
  this.firstName = firstName;
  this.birthYear = birthYear;

  this.calcAge = function () {
    console.log(2037 - this.birthYear);
  };
}

jonas.calcAge(); // 46
```

This works, but it's inefficient. Each object created with the `Person` constructor will have its own `calcAge` function, which is redundant and wasteful.

Next Step: Prototypes and Prototypal Inheritance

To avoid duplicating methods, we can use **prototypes**. This is the recommended approach, and it allows us to define methods in a way that all instances share them without having a separate copy for each.

Key Takeaways:

1. **Constructor functions** are used to create objects.
2. **The `new` operator** does four things: creates a new object, sets `this` to the new object, links the object to the constructor's prototype, and returns the new object.
3. You can create multiple instances using the same constructor function.
4. Use the `instanceof` operator to check if an object is an instance of a constructor.
5. Avoid defining methods inside constructor functions; instead, use **prototypes** for shared methods.

Understanding Prototypes in JavaScript

Prototypes in JavaScript can seem tricky, but once you grasp how they work, you'll feel like a pro developer. Let's break down the concepts in a clear and structured way.

What Are Prototypes?

1. **Every Function Has a Prototype Property**
   In JavaScript, all functions, including **constructor functions**, have a special property called `prototype`. This property is an object.

2. **Inheritance via Constructor Functions**
   When an object is created using a constructor function, it **inherits** all properties and methods that are defined on the constructor's `prototype` object. This is what we call **prototypal inheritance**.

3. **Prototype Property in Action**
   Let's say we have a constructor function `Person`:

```
function Person(firstName, birthYear) {
  this.firstName = firstName;
  this.birthYear = birthYear;
}
```

   We can now add methods to this constructor's prototype. For example:

```
Person.prototype.calcAge = function () {
  console.log(2024 - this.birthYear);
};
```

This makes `calcAge` available to all instances created by the `Person` constructor without duplicating the function across objects.

4. **Prototype Chain**
   The objects created using the `Person` constructor can access the `calcAge` method through **prototypal inheritance** even though this method is not directly on the object itself:

```
const jonas = new Person("Jonas", 1991);
jonas.calcAge(); // 33
```

When you check `jonas`:

```
console.log(jonas);
```

You'll only see its own properties (`firstName`, `birthYear`), but the method `calcAge` is available via the **prototype chain**.

Key Prototype Concepts

1. **Prototype Object (`__proto__`)**
   Each object has an internal link called `__proto__` that points to the **prototype** of the constructor function that created it. For `jonas`, this would be:

```
console.log(jonas.__proto__); // Person.prototype
```

2. **Checking the Prototype**
   You can check whether a specific prototype is part of an object's prototype chain using:

```
console.log(Person.prototype.isPrototypeOf(jonas)); // true
```

3. **Prototype Confusion**
   It's easy to confuse `Person.prototype` as the prototype of `Person`, but it's actually the prototype of objects created by `Person`. In contrast:

```
console.log(Person.prototype.isPrototypeOf(Person)); // false
```

Adding Properties to the Prototype

You can add not only methods but also properties to a prototype, making them available to all instances:

```
Person.prototype.species = "Homo Sapiens";
```

Now, all instances (e.g., `jonas`, `matilda`) will have access to this shared property:

```
console.log(jonas.species); // Homo Sapiens
```

Own Properties vs Inherited Properties

1. **Own Properties**: These are properties directly defined on an object (like `firstName` and `birthYear`).
2. **Inherited Properties**: These are properties that come from the object's prototype (like `calcAge` and `species`).

You can check if a property is an **own property** using:

```
console.log(jonas.hasOwnProperty("firstName")); // true
console.log(jonas.hasOwnProperty("species")); // false
```

Summary & Key Takeaways

- **Prototypal inheritance** allows objects to inherit methods and properties from a constructor's prototype.
- **Prototype property (`Person.prototype`)** is where shared methods and properties are stored.
- **Prototype chain (`__proto__`)** links an object to its constructor's prototype.
- **Own properties** belong directly to the object, while **inherited properties** come from the prototype.

Prototypes make JavaScript more memory-efficient by ensuring methods are not duplicated across objects but shared through the prototype chain. Let's break down the concepts from the video and visualize the key relationships in JavaScript's prototype system and constructor functions:

JavaScript Prototype Chain Explained (Diagram Overview)

1. **Person Constructor Function**
    - The constructor function `Person` creates objects.
    - When an object is created using the `new` operator, a series of steps are triggered:
        - A new empty object is created.
        - The `this` keyword inside the constructor refers to the newly created object.
        - Properties (like `name` and `birthYear`) are assigned to this object.
        - The new object is linked to `Person.prototype` using the `__proto__` property.
        - Finally, the new object is returned.
2. **Prototype of Person Constructor**

- `Person.prototype` is an object.
- It contains methods like `calcAge` that are shared among all instances created by the `Person` constructor.
- `Person.prototype` also has a property called `constructor`, which points back to `Person`. This keeps the relationship intact.

3. **Instance Creation with `new`**
   - When we create an object, say `jonas`, using `new Person()`, the new object is linked to `Person.prototype`.
   - This means that if a property or method (like `calcAge`) is not found directly on `jonas`, JavaScript will look for it in `Person.prototype`.
4. **Prototypal Inheritance and Delegation**
   - The `jonas` object does not have the `calcAge` method itself. Instead, it inherits the method from its prototype (`Person.prototype`).
   - This is how prototypal inheritance works: objects can delegate functionality to their prototypes.
5. **Prototype Chain**
   - The `__proto__` property of `jonas` points to `Person.prototype`.
   - `Person.prototype` is also an object, and its prototype is `Object.prototype`.
   - `Object.prototype` is at the top of the prototype chain, and its `__proto__` property points to `null`, marking the end of the chain.
6. **Method Lookup**
   - When we call a method on `jonas` (like `calcAge`), JavaScript first looks for it on `jonas` itself. If it's not found, it looks up the prototype chain:
     - First, in `Person.prototype`.
     - Then, in `Object.prototype` if necessary.
7. **Built-in Methods from `Object.prototype`**
   - Methods like `hasOwnProperty` are available in `Object.prototype`. If `jonas` or `Person.prototype` doesn't have them, JavaScript looks further up the chain and finds them in `Object.prototype`.

Visual Diagram Representation (Description)

1. **Person Constructor Function**
   - `Person`
     → **Person.prototype**
     → **Methods**: `calcAge`
     → **constructor**: Points to `Person`
2. **Object Creation**

- `jonas` (created by `new Person()`)
  → **__proto__: Points to `Person.prototype`
  → Looks up `calcAge` in `Person.prototype`
  → If not found, looks up in `Object.prototype`

3. **Prototype Chain**

- `jonas`
  → `Person.prototype`
  → `Object.prototype`
  → `null` (end of the chain)

Key Takeaways:

- **Prototypal Inheritance**: Objects delegate properties and methods to their prototypes.
- **Prototype Chain**: It connects objects to their prototypes, allowing method lookup across multiple levels.
- **Constructor Functions**: They create objects and link them to shared prototypes using the `__proto__` property.
- **Code Efficiency**: Methods are shared via prototypes, which prevents duplication in each object, improving performance.
- **Inheritance**: This setup allows for method sharing and efficient memory use, forming the basis for class-like inheritance in JavaScript.

Prototypal Inheritance and the Prototype Chain

Prototypal inheritance in JavaScript allows objects to inherit properties and methods from other objects. This is achieved via the `[[Prototype]]` (commonly accessible via the `__proto__` property) and the `prototype` property of constructor functions.

Let's explore the concepts using built-in objects like arrays, as well as look into how the prototype chain works for different types of objects.

Key Concepts

1. **Prototype and `__proto__`**

- Every JavaScript object has a hidden property called `[[Prototype]]`, accessible via `__proto__`, which points to another object (its prototype).
- When accessing a property or method on an object, JavaScript first checks if the object itself has it. If not, it looks up the prototype chain.

2. **Prototype of `person` Example**

- Assume you have an object `jonas` created from a constructor `Person`.
- The `__proto__` of `jonas` is the prototype of `Person`, which holds methods like `calcAge` that we defined earlier.

- The `__proto__` of `jonas.__proto__` (i.e., the prototype of `Person.prototype`) is the `Object.prototype`, which contains methods like `hasOwnProperty`.

```
console.log(jonas.__proto__); // Person.prototype
console.log(jonas.__proto__.__proto__); // Object.prototype
console.log(jonas.__proto__.__proto__.__proto__); // null
```

3. **Prototype of Built-in Objects**
   - Arrays, like objects, have a prototype. The prototype of an array contains all array methods such as `filter`, `map`, `reduce`, etc. These methods are inherited by all arrays.

```
const arr = [1, 2, 3];
console.log(arr.__proto__); // Array.prototype
console.log(arr.__proto__.__proto__); // Object.prototype
```

4. **The Prototype Chain of Arrays**
   - The array inherits from `Array.prototype`, which provides all the array-specific methods. However, `Array.prototype` itself inherits from `Object.prototype`, which contains more generic object methods like `hasOwnProperty`.
   - The chain goes:
     - `arr.__proto__` (Array methods) → `Object.prototype` (Object methods) → `null`.

Extending Built-in Prototypes (Not Recommended)

You can extend the prototype of built-in objects like arrays to add new functionality. For example, let's add a method to return unique elements in an array.

```
Array.prototype.unique = function () {
  return [...new Set(this)];
};

const arr = [1, 2, 3, 3, 4];
console.log(arr.unique()); // [1, 2, 3, 4]
```

However, **extending built-in prototypes is generally discouraged** because:

- It can lead to conflicts if future versions of JavaScript introduce a method with the same name.
- In a team environment, it can lead to confusion or conflicts if multiple developers modify prototypes.

Prototype Chain of Functions

Functions are also objects in JavaScript, and therefore, they have their own prototype. Common methods like `call`, `apply`, and `bind` are stored on the prototype of functions.

```
function example() {}
console.log(example.__proto__); // Function.prototype
console.log(example.__proto__.__proto__); // Object.prototype
```

Exploring the DOM and Prototypes

Even DOM elements in JavaScript are objects and follow the prototype chain. For instance, an `h1` element inherits from `HTMLHeadingElement`, which itself inherits from `HTMLElement`, and so on until `Node` and `Object`.

```
const h1 = document.querySelector("h1");
console.dir(h1); // Prototype chain: HTMLHeadingElement -> HTMLElement -> Element
-> Node -> EventTarget -> Object
```

The chain for DOM elements is typically long, showcasing the hierarchical structure of different types of elements.

Summary

- **Prototypal inheritance** is a powerful mechanism that allows objects to reuse properties and methods from other objects.
- **Prototype chain**: When looking for a property, JavaScript walks up the chain until it finds the property or reaches `null`.
- **Avoid extending built-in prototypes** in large projects, as it can lead to conflicts and unexpected bugs.
- Even **functions and DOM elements** have prototypes, and they follow the same prototype inheritance mechanism. Great! Let's break down the solution for the coding challenge regarding creating a simple `Car` constructor function and implementing the `accelerate` and `brake` methods.

Step-by-Step Solution

1. **Define the Constructor Function**: We'll create a constructor function called `Car` that initializes the `make` and `speed` properties.

2. **Add Methods to the Prototype**: We'll add the `accelerate` and `brake` methods to the `Car` prototype to allow all instances of `Car` to use these methods without creating duplicates.

3. **Create Car Objects**: We'll create two instances of `Car`, one for BMW and another for Mercedes, and test the methods.

Here’s how you can implement it:

Code Implementation

```
// Step 1: Define the Constructor Function
function Car(make, speed) {
  this.make = make; // Initialize the make property
  this.speed = speed; // Initialize the speed property
}

// Step 2: Add Methods to the Prototype
Car.prototype.accelerate = function () {
  this.speed += 10; // Increase speed by 10
  console.log(`${this.make} is going at ${this.speed} kilometers per hour.`);
};

Car.prototype.brake = function () {
  this.speed -= 5; // Decrease speed by 5
  console.log(`${this.make} is going at ${this.speed} kilometers per hour.`);
};

// Step 3: Create Car Objects
const bmw = new Car("BMW", 120); // Create a BMW car object
const mercedes = new Car("Mercedes", 95); // Create a Mercedes car object

// Test the methods
bmw.accelerate(); // BMW is going at 130 kilometers per hour.
bmw.accelerate(); // BMW is going at 140 kilometers per hour.
bmw.accelerate(); // BMW is going at 150 kilometers per hour.
bmw.brake(); // BMW is going at 145 kilometers per hour.

mercedes.accelerate(); // Mercedes is going at 105 kilometers per hour.
mercedes.brake(); // Mercedes is going at 100 kilometers per hour.
```

Explanation

1. **Constructor Function**: The `Car` function acts as a blueprint for creating car objects. It takes `make` and `speed` as parameters and initializes the properties accordingly.
2. **Prototype Methods**:
   - `accelerate()`: This method increases the `speed` property by 10 and logs the new speed along with the car's make to the console.
   - `brake()`: This method decreases the `speed` property by 5 and logs the new speed similarly.
3. **Creating Objects**: We create two car objects using the `new` keyword, passing the make and initial speed.
4. **Testing**: The methods are tested by calling them on each car object, and the results are logged to the console.

Conclusion

This implementation showcases the principles of object-oriented programming by encapsulating data and functionality within car objects. Each car can manage its own state (speed) and behavior (accelerate and

brake), promoting code reusability and organization.

Understanding ES6 Classes in JavaScript

In this section, we delve into the implementation of prototypal inheritance in JavaScript using ES6 classes, which offer a modern and cleaner syntax compared to constructor functions.

Key Points About Classes

1. **Syntactic Sugar**:
   - ES6 classes are essentially syntactic sugar over traditional prototypal inheritance. They simplify the syntax, making it easier for developers transitioning from other object-oriented languages like Java or C++.
2. **Class Declaration vs. Class Expression**:
   - **Class Declaration**:

```
class PersonCl {
  constructor(firstName, birthYear) {
    this.firstName = firstName;
    this.birthYear = birthYear;
  }
  // Methods can be added here
}
```

   - **Class Expression**:

```
const PersonCl = class {
  constructor(firstName, birthYear) {
    this.firstName = firstName;
    this.birthYear = birthYear;
  }
  // Methods can be added here
};
```

3. **Constructor Method**:
   - Each class must have a `constructor` method. This method is automatically called when a new instance of the class is created using the `new` keyword. It initializes properties using parameters passed to it.
   - Example of creating an instance:

```
const jessica = new PersonCl("Jessica", 1990);
```

4. **Adding Methods**:
    - Methods can be defined directly within the class body. These methods are added to the prototype of the class, allowing for inheritance.
    - Example:

```
class PersonCl {
  // ...
  calcAge() {
    console.log(2024 - this.birthYear);
  }
}
```

5. **Prototype Inheritance**:
    - Even with the class syntax, methods defined in a class are still prototype methods, which means they are not stored in the instance itself but rather on the class's prototype.
    - You can confirm this by checking the prototype of an instance:

```
console.log(jessica.__proto__ === PersonCl.prototype); // true
```

6. **Manual Prototype Methods**:
    - You can still manually add methods to the prototype of a class:

```
PersonCl.prototype.greet = function () {
  console.log(`Hey, ${this.firstName}`);
};
```

Important Notes About Classes

1. **Non-Hoisting**:
    - Unlike function declarations, class declarations are not hoisted. You cannot use a class before it is declared.
2. **First-Class Citizens**:
    - Classes can be passed as arguments to functions and returned from functions because they are special types of functions.
3. **Strict Mode**:
    - The body of a class is always executed in strict mode, enforcing stricter parsing and error handling.

Should You Use Classes or Constructor Functions?

- **Personal Preference**:
  - Constructor functions are not deprecated, and using them is perfectly fine. The choice between classes and constructor functions comes down to personal preference.
- **Understanding Prototypal Inheritance**:
  - It is essential to understand prototypal inheritance and how it works before using classes. Classes abstract this complexity but do not eliminate it.
- **Visual Clarity**:
  - Classes provide better organization of related data and behavior, making the code easier to read and maintain, especially in larger applications. Here's a structured note capturing the key points from the lecture on getters and setters in JavaScript:

Getters and Setters in JavaScript

Overview

- **Getters and Setters**: Special properties in JavaScript that allow for the retrieval and assignment of values, resembling regular properties while being implemented as functions.
- **Data Properties vs. Accessor Properties**:
  - **Data Properties**: Regular properties holding values.
  - **Accessor Properties**: Properties that are defined with getters and setters.

Creating Getters and Setters

Example: Bank Account Object

1. **Object Literal**:

```
const account = {
  owner: "Jonas",
  movements: [200, 450, -400, 300],
};
```

2. **Adding a Getter**:
   - Define a method and prepend it with the `get` keyword to create a getter.
   - Example to get the latest movement:

```
get latest() {
    return this.movements.slice(-1).pop();
}
```

- Usage:

```
console.log(account.latest); // Returns the last movement
```

3. **Adding a Setter**:
   - Define a method with the `set` keyword.
   - Example to add a new movement:

```
set latest(movement) {
    this.movements.push(movement);
}
```

   - Usage:

```
account.latest = 50; // Adds 50 to movements
```

Getters and Setters in Classes

Example: Person Class

1. **Class Definition**:

```
class Person {
  constructor(firstName, birthYear) {
    this.firstName = firstName;
    this.birthYear = birthYear;
  }

  // Getter for age
  get age() {
    return new Date().getFullYear() - this.birthYear;
  }
}
```

2. **Using the Getter**:

```
const jessica = new Person("Jessica", 1990);
console.log(jessica.age); // Returns age calculated from birth year
```

Data Validation with Setters

1. **Changing Property Name**:
    - Change `firstName` to `fullName` for validation.

```
set fullName(name) {
    if (name.includes(' ')) {
        this._fullName = name; // Use an underscore to avoid naming conflict
    } else {
        alert('The given name is not a full name.');
    }
}
```

2. **Getter for `fullName`**:

```
get fullName() {
    return this._fullName;
}
```

3. **Usage**:

```
const jessica = new Person("Jessica", 1990);
jessica.fullName = "Jessica Davis"; // Valid input
console.log(jessica.fullName); // Returns 'Jessica Davis'

jessica.fullName = "Jessica"; // Invalid input, triggers alert
```

Important Notes

- **Naming Convention**: When using a setter for a property that already exists, it is common practice to use an underscore (e.g., `_fullName`) to avoid conflicts.
- **Use Cases**: Getters and setters can be particularly useful for data validation and computed properties, allowing for clean and intuitive object interfaces.

Conclusion

- Getters and setters are powerful tools in JavaScript for managing object properties, offering both flexibility and control over how values are accessed and mutated. Here's a structured summary of the video on static methods in JavaScript, highlighting key concepts, examples, and takeaways.

Static Methods in JavaScript

Overview

- Static methods are functions that are attached to the constructor itself rather than the instances of the constructor.
- They are often used as helper functions related to the constructor.

Example: `Array.from`

- `Array.from` is a built-in static method that converts array-like structures into real arrays.
- **Usage in the Console**:

```
const elements = document.querySelectorAll("div"); // This returns a
NodeList
const array = Array.from(elements); // Converts NodeList to an array
```

- If you attempt to call `from` on an array instance (e.g., `someArray.from()`), it will result in an error because `from` is not a method on the array's prototype.

Characteristics of Static Methods

- **Namespace**: Static methods are in the constructor's namespace. For instance, `Number.parseFloat` is another example of a static method, not available on number instances.
- **Not Inherited**: Static methods are not accessible from instances. For example, you cannot call `Jonas.hey()` if `hey` is defined as a static method on the `Person` constructor.

Implementing Static Methods

1. Using a Constructor Function

- Example of adding a static method:

```
function Person() {
  // constructor logic
}

Person.hey = function () {
  console.log("Hey there! 👋");
};

// Call the static method
Person.hey(); // Outputs: "Hey there! 👋"
```

- The `this` keyword refers to the constructor function itself, not an instance of the object.

2. Using a Class

- In a class, defining a static method is straightforward using the `static` keyword:

```
class PersonClass {
  static hey() {
    console.log("Hey there! 👋");
  }
}

// Call the static method
PersonClass.hey(); // Outputs: "Hey there! 👋"
```

- The `this` keyword inside a static method points to the class itself.

Key Differences

- **Static Methods**:
  - Attached to the constructor.
  - Not available on instances.
  - Useful for helper functions related to the class/constructor.
- **Instance Methods**:
  - Attached to the prototype.
  - Available to instances created from the constructor.

Conclusion

Static methods provide a way to group functionality that is related to a class or constructor, but not dependent on individual instances. They help organize code and can simplify access to certain utilities related to the class or constructor.

Summary of Object.create and Prototypal Inheritance

In this lesson, we explored a third way to implement prototypal inheritance using `Object.create`, which differs significantly from the previous methods that use constructor functions or ES6 classes.

Key Concepts

1. **Object.create**:
   - It is used to create a new object and explicitly set its prototype to another object. Unlike constructor functions and classes, there are no prototype properties involved, and the `new` operator is not used.
2. **Manual Prototype Assignment**:
   - With `Object.create`, we can manually assign the prototype of an object to another object. This allows for more control over inheritance without relying on automatic prototype assignment as in constructor functions or classes.
3. **Creating a Prototype Object**:

- We first create an object (`PersonProto`) that contains methods like `calcAge`. This object serves as the prototype for any new objects created using `Object.create`.

4. **Creating an Object Using Object.create**:
   - We can create a new object (e.g., `steven`) by passing the prototype object (`PersonProto`) to `Object.create`. The new object will inherit the methods from `PersonProto`.

```
const steven = Object.create(PersonProto);
```

5. **Setting Properties**:
   - Properties can be manually set on the new object, but this is not an ideal way to initialize objects.

```
steven.firstName = "Steven";
steven.birthYear = 1985;
```

6. **Adding an Initialization Method**:
   - To avoid manually setting properties, we can create an `init` method within `PersonProto` that acts similarly to a constructor. This method allows us to initialize properties programmatically.

```
PersonProto.init = function (firstName, birthYear) {
  this.firstName = firstName;
  this.birthYear = birthYear;
};

const sarah = Object.create(PersonProto);
sarah.init("Sarah", 1979);
```

7. **Prototype Chain**:
   - The prototype chain works the same way as with constructor functions and classes. When a method is called on an object, JavaScript first looks for the method on the object itself and, if not found, moves up the prototype chain.

Important Takeaways

- `Object.create` provides a way to directly link objects to prototypes without the use of constructor functions or classes.
- Methods are inherited through the prototype chain just like with constructor functions or classes, but the process is more manual and explicit.
- `Object.create` is less commonly used in practice, but it's still important to understand as it can be useful for certain use cases and is a key concept in JavaScript.

Example Code

```
// Create the prototype object
const PersonProto = {
  calcAge() {
    console.log(2024 - this.birthYear);
  },

  init(firstName, birthYear) {
    this.firstName = firstName;
    this.birthYear = birthYear;
  },
};

// Create a new object with PersonProto as its prototype
const sarah = Object.create(PersonProto);
sarah.init("Sarah", 1979);

// Access inherited method
sarah.calcAge(); // Output: 45
```

Great! Let's walk through how to solve this coding challenge by using an ES6 class. The goal is to recreate a car object using ES6 classes, and then add a getter and setter to handle the car's speed in miles per hour.

Step-by-Step Solution:

1. **Create the Class:** We'll define the `Car` class with a constructor to initialize the car's make and speed in kilometers per hour.
2. **Add Methods:** The class will also include methods to accelerate and brake the car by increasing or decreasing the speed respectively.
3. **Add Getter and Setter for `speedUS`:**
    - **Getter:** This will return the speed in miles per hour (mph). Since the speed is stored in kilometers per hour (km/h), we need to convert it to mph by dividing by 1.6.
    - **Setter:** This will allow setting the speed in miles per hour. When setting the speed in mph, we need to convert it back to km/h by multiplying the value by 1.6.

Code Implementation:

```
// 1. Define the Car class
class Car {
  constructor(make, speed) {
    this.make = make;
    this.speed = speed; // Speed in kilometers per hour
  }

  // 2. Method to accelerate the car
  accelerate() {
    this.speed += 10;
```

```
    console.log(`${this.make} is going at ${this.speed} km/h`);
  }

  // 3. Method to brake the car
  brake() {
    this.speed -= 5;
    console.log(`${this.make} is going at ${this.speed} km/h`);
  }

  // 4. Getter to get speed in miles per hour (mph)
  get speedUS() {
    return this.speed / 1.6;
  }

  // 5. Setter to set speed in miles per hour (mph)
  set speedUS(speed) {
    this.speed = speed * 1.6; // Convert mph to km/h
  }
}

// 6. Create a new car object
const ford = new Car("Ford", 120);

// 7. Test the getter
console.log(ford.speedUS); // Should print speed in mph (120 km/h is 75 mph)

// 8. Test the setter
ford.speedUS = 50; // Setting speed to 50 mph
console.log(ford.speed); // Should print 80 km/h (50 mph is 80 km/h)

// 9. Accelerate and brake the car
ford.accelerate(); // Increases speed by 10 km/h
ford.brake(); // Decreases speed by 5 km/h
```

Explanation:

- **Constructor:** The `constructor` method takes two arguments: the `make` of the car and the `speed` in kilometers per hour.
- **Methods (`accelerate`, `brake`):** These methods modify the car's speed. `accelerate()` increases the speed by 10 km/h, and `brake()` decreases it by 5 km/h.
- **Getter `speedUS`:** The getter converts the car's speed from kilometers per hour to miles per hour. We divide by 1.6 to perform the conversion.
- **Setter `speedUS`:** The setter allows you to set the car's speed in miles per hour. It multiplies the input speed (mph) by 1.6 to convert it back to kilometers per hour and updates the `speed` property accordingly.

Test Data:

- When you create a `Car` object like `ford` with a speed of 120 km/h, the `speedUS` getter should return 75 mph.
- If you set `speedUS = 50`, it will set the car's speed to 80 km/h (since 50 mph × 1.6 = 80 km/h). In this series of lectures, we've been learning about prototypal inheritance in JavaScript, using various techniques like constructor functions, ES6 classes, and `Object.create`. Now we're transitioning to understanding "real" inheritance between classes, focusing on inheriting properties and methods across parent and child classes, specifically creating a `Student` class that inherits from a `Person` class. Here's a structured summary of the key concepts and implementation steps covered so far:

Key Concepts:

1. **Prototypal Inheritance Recap**:
    - Objects delegate behavior (methods) to their prototype.
    - Inheritance in JavaScript doesn't use "real" classes like other languages but rather prototypes.
2. **Class-Based Inheritance**:
    - **Parent Class**: `Person`, with methods like `calcAge`.
    - **Child Class**: `Student`, inheriting from `Person` but with its own properties and methods (like `course` and `introduce`).
    - Inheritance allows the `Student` class to use methods from `Person`, such as `calcAge`, alongside its own methods.
3. **Techniques of Inheritance**:
    - **Constructor Functions**: We're creating inheritance between constructor functions manually.
    - **Object.create**: Used to link the prototype of the child to the prototype of the parent.

Steps to Create Inheritance in Constructor Functions:

1. **Defining the Parent (`Person`) Constructor**:

```
const Person = function (firstName, birthYear) {
  this.firstName = firstName;
  this.birthYear = birthYear;
};

Person.prototype.calcAge = function () {
  console.log(2024 - this.birthYear);
};
```

2. **Defining the Child (`Student`) Constructor**:

```
const Student = function (firstName, birthYear, course) {
  // Call the Person constructor to initialize firstName and birthYear
  Person.call(this, firstName, birthYear);
```

```
  this.course = course;
};
```

- **`call` method**: Used to invoke the `Person` constructor within `Student` and correctly set the `this` context to ensure the properties are initialized on the newly created `Student` instance.

3. **Linking the Prototypes**:

```
Student.prototype = Object.create(Person.prototype);
```

- This is crucial to set up the prototype chain so that methods in `Person.prototype` are accessible from `Student` instances.
- `Object.create` creates a new object that inherits from `Person.prototype`.

4. **Adding Methods to `Student.prototype`**:

```
Student.prototype.introduce = function () {
  console.log(`My name is ${this.firstName} and I study ${this.course}`);
};
```

5. **Fixing the Constructor Reference**: After using `Object.create`, the `constructor` of `Student.prototype` points to `Person`, so we need to fix that:

```
Student.prototype.constructor = Student;
```

Example Usage:

```
const mike = new Student("Mike", 2020, "Computer Science");
mike.introduce(); // My name is Mike and I study Computer Science
mike.calcAge(); // 4 (since it's 2024)
```

- **`mike.introduce()`**: This calls the `introduce` method defined on `Student.prototype`.
- **`mike.calcAge()`**: This method is inherited from `Person.prototype` through the prototype chain.

Understanding the Prototype Chain:

- When `mike.introduce()` is called, JavaScript looks for the method in `Student.prototype`.
- When `mike.calcAge()` is called, JavaScript looks up the prototype chain and finds the method in `Person.prototype`.

- The `Student` objects inherit all methods defined on the `Person.prototype`, thanks to the linked prototype.

Why `Object.create` and Not Direct Assignment?:

- Directly setting `Student.prototype = Person.prototype` would make both prototypes the **same object**, meaning any changes to one would affect the other, which we don't want.
- Using `Object.create(Person.prototype)` ensures `Student.prototype` is a new object that inherits from `Person.prototype` without being the same object.

---

221 lecture
