JavaScript Fundamentals Part 1

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.

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.

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.