Fundamentals of Modern JavaScript - ES6 and Beyond: Revised Course Material

This course material is designed for instructors to deliver *Fundamentals of Modern JavaScript - ES6 and Beyond* over weekly sessions. It covers three modules with detailed explanations, step-by-step breakdowns of examples, and real-world applications with solutions, progressing from basic to advanced concepts to meet the specified learning outcomes.

Module 1: Datatypes and Variables

Overview

This module introduces JavaScript's core datatypes and variable declarations, focusing on understanding primitive and reference types, manipulating variables, and applying built-in methods. Each concept is explained with detailed examples and real-world scenarios.

Learning Outcomes

- Identify and differentiate between primitive and reference data types.
- Demonstrate proficiency in declaring and initializing variables.
- Perform basic operations on variables (assignment, arithmetic, string concatenation, comparison).
- Utilize built-in JavaScript methods for strings and arrays.
- Understand scope and hoisting.

Content

1.1 JavaScript Datatypes

JavaScript datatypes are divided into **primitive** and **reference** types. Primitive types are simple, immutable values stored directly in memory, while reference types are complex and stored as pointers to memory locations.

- **Primitive Types**:
  - `number`: Represents integers or decimals (e.g., `42`, `3.14`).
  - `string`: Text enclosed in single or double quotes (e.g., `"hello"`).
  - `boolean`: Logical values `true` or `false`.
  - `undefined`: A variable declared but not assigned a value.
  - `null`: Intentional absence of a value.
  - `symbol` (ES6): Unique identifiers, often used for object properties.
  - `bigint`: For integers beyond the `number` type's limit.
- **Reference Types**:
  - `object`: Collections of key-value pairs.
  - `array`: Ordered lists of values.
  - `function`: Reusable code blocks.

Example with Explanation:

```
let age = 25; // number: Stores the integer 25
let name = "Alice"; // string: Stores the text "Alice"
let isStudent = true; // boolean: Represents a true/false state
let noValue = undefined; // undefined: Declared but unassigned
let empty = null; // null: Explicitly no value
let id = Symbol("id"); // symbol: Unique identifier for "id"
let bigNumber = 12345678901234567890n; // bigint: Large integer
let person = { name: "Bob", age: 30 }; // object: Stores key-value pairs
let numbers = [1, 2, 3]; // array: Ordered list
let greet = function() { console.log("Hello"); }; // function: Executable
code
```

Explanation:

- `age = 25`: Assigns the number `25` to the variable `age`. Numbers are used for calculations like age or price.
- `name = "Alice"`: Stores the string `"Alice"`. Strings are used for text data like names or messages.
- `isStudent = true`: A boolean to represent a condition, useful for toggling states (e.g., is the user a student?).
- `noValue = undefined`: Indicates a variable exists but has no value yet, often a default state.
- `empty = null`: Explicitly indicates no value, used to reset or clear data.
- `id = Symbol("id")`: Creates a unique identifier, useful for preventing property name conflicts in objects.
- `bigNumber = 12345678901234567890n`: Handles very large integers, like financial calculations.
- `person = { name: "Bob", age: 30 }`: An object storing multiple related values, like a user profile.
- `numbers = [1, 2, 3]`: An array for lists, like a collection of items.
- `greet = function() { ... }`: A function to execute code, like displaying a message.

Real-World Example with Solution: In a social media app, you need to store a user's profile data.

```
// Solution: Create a user profile with mixed datatypes
let userProfile = {
  username: "alice123", // string: User's display name
  followers: 150, // number: Count of followers
  isActive: true, // boolean: Account status
  posts: ["Post 1", "Post 2"], // array: List of posts
  uniqueId: Symbol("userId") // symbol: Unique identifier
};
console.log(userProfile.username); // "alice123"
console.log(userProfile.posts[0]); // "Post 1"
```

Explanation of Solution:

- The `userProfile` object combines multiple datatypes to represent a user.
- `username` is a string for the user's name, displayed on their profile.
- `followers` is a number for tracking social metrics.
- `isActive` is a boolean to check if the account is active.
- `posts` is an array for storing user posts, accessible by index.
- `uniqueId` is a symbol to ensure a unique property key.

1.2 Variable Declarations

JavaScript offers three ways to declare variables:

- `var`: Function-scoped, can be redeclared, hoisted with `undefined`.
- `let`: Block-scoped, cannot be redeclared in the same scope, hoisted but not initialized.
- `const`: Block-scoped, cannot be reassigned (but objects/arrays are mutable), hoisted but not initialized.

Example with Explanation:

```
var oldStyle = "Old way"; // Can be redeclared
var oldStyle = "New value"; // No error
let modern = "Modern way"; // Block-scoped
// let modern = "Error"; // Error: Cannot redeclare
const fixed = "Cannot change"; // Constant
// fixed = "Error"; // Error: Cannot reassign
const obj = { value: 10 };
obj.value = 20; // Allowed: Object properties are mutable
```

Explanation:

- `var oldStyle`: Declares a variable that can be redeclared. It's hoisted, so it's accessible before its declaration but as `undefined`.
- `let modern`: Declares a block-scoped variable. Attempting to redeclare `modern` in the same scope causes an error.
- `const fixed`: Declares a constant that cannot be reassigned. However, for objects like `obj`, the object's properties can still be modified.

Real-World Example with Solution: In an e-commerce app, store a product's price and update its discount.

```
// Solution
const product = {
  name: "Laptop",
  price: 1000
};
let discount = 0.1; // 10% discount
product.price = product.price * (1 - discount); // Update price
console.log(product.price); // 900
```

Explanation of Solution:

- `product` is a `const` object, meaning the reference cannot change, but its `price` property can be updated.
- `discount` is a `let` variable, allowing reassignment if the discount changes.
- The price is updated by applying a 10% discount, simulating a sale calculation.

1.3 Manipulating Strings and Arrays

JavaScript provides built-in methods to manipulate strings and arrays:

- **String Methods**:
    - `toUpperCase()`: Converts string to uppercase.
    - `slice(start, end)`: Extracts a substring.
    - `replace(search, replacement)`: Replaces part of the string.
- **Array Methods**:
    - `push()`: Adds an element to the end.
    - `pop()`: Removes the last element.
    - `map(callback)`: Transforms each element.

Example with Explanation:

```
let text = "Hello, World!";
let upperText = text.toUpperCase(); // "HELLO, WORLD!"
let sliced = text.slice(0, 5); // "Hello"
let replaced = text.replace("World", "JavaScript"); // "Hello, JavaScript!"

let arr = [1, 2, 3];
arr.push(4); // [1, 2, 3, 4]
let slicedArr = arr.slice(1, 3); // [2, 3]
let doubled = arr.map(x => x * 2); // [2, 4, 6, 8]
```

Explanation:

- `toUpperCase()`: Converts `"Hello, World!"` to all uppercase, useful for standardizing text.
- `slice(0, 5)`: Extracts characters from index 0 to 4, returning `"Hello"`.
- `replace("World", "JavaScript")`: Replaces `"World"` with `"JavaScript"`, useful for dynamic text updates.
- `push(4)`: Adds `4` to the array's end, modifying it in place.
- `slice(1, 3)`: Returns a new array with elements from index 1 to 2 (`[2, 3]`).
- `map(x => x * 2)`: Creates a new array where each element is doubled.

Real-World Example with Solution: In a blog app, format a post's title and tag list.

```
// Solution
let postTitle = "learn javascript now";
let tags = ["coding", "javascript"];
postTitle = postTitle.toUpperCase(); // "LEARN JAVASCRIPT NOW"
tags.push("webdev"); // Add new tag
```

```
let tagDisplay = tags.map(tag => `#${tag}`); // ["#coding", "#javascript",
"#webdev"]
console.log(postTitle, tagDisplay);
```

Explanation of Solution:

- `toUpperCase()`: Converts the title to uppercase for emphasis in a UI.
- `push("webdev")`: Adds a new tag to the array, simulating adding a category.
- `map(tag => #${tag})`: Adds a `#` prefix to each tag, creating a hashtag format for display.

1.4 Scope and Hoisting

- **Scope**: Determines where variables are accessible.
  - **Global Scope**: Variables declared outside functions, accessible everywhere.
  - **Local Scope**: Variables inside functions or blocks, accessible only there.
- **Hoisting**: JavaScript moves declarations (not initializations) to the top of their scope during compilation.

Example with Explanation:

```
console.log(x); // undefined (hoisted, but not initialized)
var x = 5;

function testScope() {
  let localVar = "I am local";
  console.log(localVar); // "I am local"
}
testScope();
console.log(localVar); // Error: localVar is not defined
```

Explanation:

- `console.log(x)`: Outputs `undefined` because `var x` is hoisted, but its assignment (`x = 5`) happens later.
- `let localVar`: Declared inside `testScope`, it's only accessible within that function (block scope).
- `console.log(localVar)` outside the function fails because `localVar` is not in the global scope.

Real-World Example with Solution: In a settings page, manage user preferences with proper scoping.

```
// Solution
let theme = "light"; // Global scope
function updatePreferences() {
  let fontSize = 16; // Local scope
  console.log(`Theme: ${theme}, Font Size: ${fontSize}`); // Access both
}
updatePreferences(); // "Theme: light, Font Size: 16"
console.log(theme); // "light"
```

```
// console.log(fontSize); // Error: fontSize is not defined
```

Explanation of Solution:

- `theme` is global, accessible everywhere, like an app-wide setting.
- `fontSize` is local to `updatePreferences`, preventing accidental access elsewhere.
- The function combines global and local variables to display user settings.

Practical Exercise

1. Create a user profile object and use string methods to format the name (e.g., capitalize first letter) and array methods to add a hobby.
2. Write a program to demonstrate hoisting with `var` and the error with `let`.

Module 2: Sequence, Selection, and Iteration

Overview

This module covers control flow structures, error handling, and algorithmic thinking, enabling students to control program execution and solve problems logically.

Learning Outcomes

- Implement sequence, selection (if/else), and iteration (loops).
- Understand conditional statements and loop syntax.
- Recognize and fix errors using debugging techniques.
- Implement try-catch for error handling.
- Apply algorithmic thinking to solve problems.

Content

2.1 Sequence

Sequence means executing statements in the order they appear.

Example with Explanation:

```
let price = 100;
let tax = price * 0.1;
let total = price + tax;
console.log(total); // 110
```

Explanation:

- `price = 100`: Sets the base price.
- `tax = price * 0.1`: Calculates 10% tax (10).
- `total = price + tax`: Adds tax to price (100 + 10 = 110).

- `console.log(total)`: Outputs the final total. Each step executes sequentially.

Real-World Example with Solution: In a restaurant app, calculate the bill with tax and tip.

```
// Solution
let mealCost = 50;
let taxRate = 0.08; // 8% tax
let tipRate = 0.15; // 15% tip
let tax = mealCost * taxRate; // 4
let tip = mealCost * tipRate; // 7.5
let totalBill = mealCost + tax + tip; // 61.5
console.log(`Total Bill: $${totalBill}`); // "Total Bill: $61.5"
```

Explanation of Solution:

- Each calculation (tax, tip, total) is performed in order.
- `tax` is 8% of $50 ($4).
- `tip` is 15% of $50 ($7.5).
- `totalBill` sums the meal cost, tax, and tip, simulating a restaurant receipt.

2.2 Selection (Conditional Statements)

Conditional statements (`if`, `else if`, `else`, `switch`) control program flow based on conditions.

Example with Explanation:

```
let age = 20;
if (age >= 18) {
  console.log("Adult");
} else {
  console.log("Minor");
}

let day = "Monday";
switch (day) {
  case "Monday":
    console.log("Start of the week");
    break;
  default:
    console.log("Another day");
}
```

Explanation:

- `if (age >= 18)`: Checks if `age` is 18 or more. Since `20 >= 18`, it logs `"Adult"`.
- `switch (day)`: Matches `day` against cases. Since `day` is `"Monday"`, it logs `"Start of the week"`. The `break` prevents fall-through to the `default` case.

Real-World Example with Solution: In a ticketing system, determine ticket price based on age.

```
// Solution
function getTicketPrice(age) {
  if (age < 13) {
    return 5; // Child ticket
  } else if (age < 65) {
    return 10; // Adult ticket
  } else {
    return 7; // Senior ticket
  }
}
console.log(`Ticket price: $${getTicketPrice(25)}`); // "Ticket price: $10"
```

Explanation of Solution:

- The function checks `age` against conditions:
  - `< 13`: Child ticket ($5).
  - `< 65`: Adult ticket ($10).
  - Otherwise: Senior ticket ($7).
- For `age = 25`, it returns $10 (adult ticket).

2.3 Iteration (Loops)

Loops (`for`, `while`, `do-while`) repeat code based on conditions.

Example with Explanation:

```
for (let i = 0; i < 3; i++) {
  console.log(i); // 0, 1, 2
}

let count = 0;
while (count < 3) {
  console.log(count);
  count++;
}
```

Explanation:

- `for (let i = 0; i < 3; i++)`: Initializes `i` to 0, runs while `i < 3`, and increments `i` each iteration. Logs 0, 1, 2.
- `while (count < 3)`: Runs as long as `count < 3`. Increments `count` and logs 0, 1, 2.

Real-World Example with Solution: In an email app, display unread messages.

```
// Solution
let messages = ["Msg 1", "Msg 2", "Msg 3"];
for (let i = 0; i < messages.length; i++) {
  console.log(`Unread: ${messages[i]}`);
}
// Output:
// Unread: Msg 1
// Unread: Msg 2
// Unread: Msg 3
```

Explanation of Solution:

- The `for` loop iterates over the `messages` array.
- `i` starts at 0 and runs until it equals `messages.length` (3).
- Each iteration logs a message with "Unread" prefix, simulating an inbox display.

2.4 Error Handling

Use `try-catch` to handle exceptions and debugging tools (e.g., `console.log`, browser DevTools) to identify errors.

Example with Explanation:

```
try {
  let result = undefinedVar; // Causes an error
} catch (error) {
  console.log("Error:", error.message); // "undefinedVar is not defined"
}
```

Explanation:

- `try`: Attempts to access `undefinedVar`, which doesn't exist.
- `catch`: Captures the error and logs its message, preventing the program from crashing.

Real-World Example with Solution: In a form, validate JSON input.

```
// Solution
function parseUserInput(input) {
  try {
    let data = JSON.parse(input);
    console.log("Valid JSON:", data);
  } catch (error) {
    console.log("Invalid JSON:", error.message);
  }
}
parseUserInput('{"name": "Alice"}'); // "Valid JSON: { name: 'Alice' }"
parseUserInput("invalid"); // "Invalid JSON: Unexpected token i in JSON at
position 0"
```

Explanation of Solution:

- `JSON.parse(input)`: Attempts to parse a string as JSON.
- If `input` is valid JSON (e.g., `'{"name": "Alice"}'`), it logs the parsed object.
- If invalid (e.g., `"invalid"`), the `catch` block logs the error message.

2.5 Algorithmic Thinking

Algorithmic thinking involves designing step-by-step solutions to problems.

Example with Explanation:

```
function sumEvenNumbers(arr) {
  let sum = 0;
  for (let num of arr) {
    if (num % 2 === 0) {
      sum += num;
    }
  }
  return sum;
}
console.log(sumEvenNumbers([1, 2, 3, 4])); // 6
```

Explanation:

- Initialize `sum = 0` to track the total.
- Loop through each `num` in `arr` using a `for...of` loop.
- Check if `num` is even (`num % 2 === 0`).
- Add even numbers to `sum`. For `[1, 2, 3, 4]`, adds `2 + 4 = 6`.

Real-World Example with Solution: In a finance app, sum even-valued transactions.

```
// Solution
function sumEvenTransactions(transactions) {
  let total = 0;
  for (let amount of transactions) {
    if (amount % 2 === 0) {
      total += amount;
    }
  }
  return total;
}
let transactions = [15, 20, 33, 40];
console.log(`Sum of even transactions:
$${sumEvenTransactions(transactions)}`); // "Sum of even transactions: $60"
```

Explanation of Solution:

- The function loops through `transactions` ([15, 20, 33, 40]).
- Checks if each `amount` is even (20 and 40 are even).
- Sums even amounts: `20 + 40 = 60`.
- Logs the result, useful for analyzing specific transaction patterns.

Practical Exercise

1. Write a program to categorize a user's age (child: <13, teen: 13-17, adult: ≥18) using `if-else`.
2. Use a `for` loop to print the first 5 Fibonacci numbers (0, 1, 1, 2, 3).

3. Implement a `try-catch` block to handle division by zero in a calculator function.

Module 3: Working with Functions in Modern JavaScript

Overview

This module focuses on functions, their scope, closures, and functional programming concepts, enabling students to write modular, reusable, and expressive code.

Learning Outcomes

- Define and invoke functions.
- Differentiate between parameters and arguments.
- Explain function scope and closures.
- Apply functional programming principles (immutability, higher-order functions, function composition).

Content

3.1 Function Declaration and Invocation

Functions are reusable code blocks that perform specific tasks. They can be declared using `function` or arrow syntax (`=>`).

Example with Explanation:

```
function greet(name) {
  return `Hello, ${name}!`;
}
console.log(greet("Alice")); // "Hello, Alice!"

const add = (a, b) => a + b;
console.log(add(2, 3)); // 5
```

Explanation:

- `function greet(name)`: Declares a function that takes a `name` parameter and returns a greeting.
- `greet("Alice")`: Invokes the function with `"Alice"` as an argument, returning `"Hello, Alice!"`.
- `const add = (a, b) => a + b`: An arrow function that adds two numbers. `add(2, 3)` returns `5`.

Real-World Example with Solution: In a notification system, generate welcome messages.

```
// Solution
const sendWelcome = (username) => `Welcome, ${username}! Enjoy our
platform.`;
console.log(sendWelcome("Bob")); // "Welcome, Bob! Enjoy our platform."
```

Explanation of Solution:

- The `sendWelcome` arrow function takes a `username` and returns a formatted message.
- Calling `sendWelcome("Bob")` generates a personalized notification for the user.

3.2 Parameters vs. Arguments

- **Parameters**: Variables in the function definition.
- **Arguments**: Values passed when calling the function. Default parameters provide fallback values.

Example with Explanation:

```
function calculateTotal(price, taxRate = 0.1) {
  return price + price * taxRate;
}
console.log(calculateTotal(100, 0.2)); // 120
console.log(calculateTotal(100)); // 110
```

Explanation:

- `price` and `taxRate` are parameters; `taxRate` defaults to `0.1` if not provided.
- `calculateTotal(100, 0.2)`: Passes `100` (price) and `0.2` (taxRate). Calculates `100 + 100 * 0.2 = 120`.
- `calculateTotal(100)`: Uses default `taxRate = 0.1`. Calculates `100 + 100 * 0.1 = 110`.

Real-World Example with Solution: In a shopping cart, calculate item totals with optional discounts.

```
// Solution
function calculateItemTotal(price, discount = 0) {
  return price * (1 - discount);
}
console.log(calculateItemTotal(200, 0.25)); // 150
console.log(calculateItemTotal(200)); // 200
```

Explanation of Solution:

- `price` and `discount` are parameters; `discount` defaults to 0.
- `calculateItemTotal(200, 0.25)`: Applies a 25% discount: `200 * (1 - 0.25) = 150`.
- `calculateItemTotal(200)`: No discount, returns `200`.

3.3 Function Scope and Closures

- **Function Scope**: Variables inside a function are local and inaccessible outside.
- **Closures**: A function that retains access to its outer scope's variables after the outer function finishes.

Example with Explanation:

```
function counter() {
  let count = 0;
  return function() {
    return ++count;
  };
}
const myCounter = counter();
console.log(myCounter()); // 1
console.log(myCounter()); // 2
```

Explanation:

- `counter` defines `count = 0` and returns an inner function.
- The inner function forms a closure, retaining access to `count`.
- `myCounter` is the inner function. Each call increments `count`: first call returns `1`, second call returns `2`.

Real-World Example with Solution: In a game, track a player's score privately.

```
// Solution
function createScoreTracker() {
  let score = 0;
  return function(points) {
    score += points;
    return `Current score: ${score}`;
  };
}
const playerScore = createScoreTracker();
console.log(playerScore(10)); // "Current score: 10"
console.log(playerScore(5)); // "Current score: 15"
```

Explanation of Solution:

- `createScoreTracker` initializes `score = 0` and returns a function that adds `points` to `score`.
- The closure ensures `score` is private and persists between calls.
- `playerScore(10)` adds 10 points, returning `"Current score: 10"`.
- `playerScore(5)` adds 5 more, returning `"Current score: 15"`.

3.4 Functional Programming Concepts

Functional programming emphasizes immutability, higher-order functions, and function composition:

- **Immutability**: Avoid modifying data; create new copies.
- **Higher-Order Functions**: Functions that accept or return functions.
- **Function Composition**: Combine functions to create new ones.

Example with Explanation:

```
const numbers = [1, 2, 3];
const doubled = numbers.map(num => num * 2); // [2, 4, 6]

const filterEven = arr => arr.filter(num => num % 2 === 0);
console.log(filterEven([1, 2, 3, 4])); // [2, 4]

const compose = (f, g) => x => f(g(x));
const addOne = x => x + 1;
const double = x => x * 2;
const addThenDouble = compose(double, addOne);
console.log(addThenDouble(5)); // 12
```

Explanation:

- `map(num => num * 2)`: Creates a new array with each element doubled, leaving `numbers` unchanged (immutability).
- `filter(num => num % 2 === 0)`: A higher-order function that returns a new array with only even numbers.
- `compose(double, addOne)`: Combines functions so `addThenDouble(5)` first applies `addOne(5)` (6), then `double(6)` (12).

Real-World Example with Solution: In a data dashboard, process sales data without mutating the original dataset.

```
// Solution
const sales = [100, 150, 200, 250];
const applyTax = num => num * 1.1; // Add 10% tax
const filterHighSales = arr => arr.filter(sale => sale > 150);
const processSales = compose(filterHighSales, sales => sales.map(applyTax));
console.log(processSales(sales)); // [220, 275]
console.log(sales); // [100, 150, 200, 250] (unchanged)
```

Explanation of Solution:

- `applyTax`: Adds 10% tax to a sale (e.g., `100 * 1.1 = 110`).
- `filterHighSales`: Filters sales above 150.
- `compose`: First applies `map(applyTax)` to get `[110, 165, 220, 275]`, then `filterHighSales` to get `[220, 275]`.
- The original `sales` array remains unchanged, demonstrating immutability.

Practical Exercise

1. Write a recursive function to calculate the factorial of a number.
2. Create a closure to manage a shopping cart's total and item count.
3. Use `map` and `filter` to process an array of products (e.g., filter by price > $50, then add 10% tax).