Intermediate JavaScript Programming

LESSON 2: ES6+ Features & Modern Syntax

Learning Objectives:

By the end of this lesson, participants will be able to:

- Use template literals to embed expressions in strings.
- Understand how arrow functions differ from traditional functions.
- Identify when to use static vs. instance methods in classes.
- Write concise and modern JavaScript using ES6+ syntax.

Lesson Outline:

I. Template Literals and String Interpolation (10 min)

- Template literals use backticks (`) instead of quotes.
- Expressions can be embedded with `${...}`.

Example:

```
const name = "Greg";
const greeting = `Hello, ${name}!`;
console.log(greeting); // Hello, Greg!
```

Exercise 1: Convert this string concatenation into a template literal.

```
const a = 5, b = 10;
console.log("The sum is: " + (a + b));
```

Answer:

```
console.log(`The sum is: ${a + b}`);
```

Short Answer: What is one advantage of using template literals over traditional string concatenation?
(Expected Answer: Easier to read and maintain, especially with multiple variables.)

II. Functions as Objects and `this` Binding (30 min)

What is a Function Object?

In JavaScript, functions are objects. That means they:

- Can be assigned to variables

- Can be passed to other functions
- Can have properties

Example:

```
function sayHi() {
  console.log("Hi!");
}

sayHi(); // Output: Hi!
sayHi.message = "Hello there!";
console.log(sayHi.message); // Output: Hello there!
```

Functions can be passed around just like any value:

```
function callTwice(func) {
  func();
  func();
}

callTwice(sayHi); // Output: Hi! Hi!
```

Behind the scenes, when you define a function:

```
function greet() {
  return "Hello";
}
```

You're creating a special kind of object — a **function object** — that has executable behavior and can also carry properties.

II. Arrow Functions and `this` Binding (continued)

- Arrow functions are a compact way to write functions, introduced in ES6.
- Syntax:

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

// Arrow function
const add = (a, b) => a + b;
```

- Arrow functions omit the `function` keyword.
- If the body is a single expression, it is returned implicitly.
- If there's only one parameter, parentheses can be omitted:

```
const square = x => x * x;
```

- If there are no parameters, use empty parentheses:

```
const sayHello = () => "Hello!";
```

What is `this`?

- `this` is a special keyword in JavaScript that refers to the object that is executing the current function.
- The object that `this` refers to is determined by **how** a function is called — this is called the `this` binding.

Examples:

```
const dog = {
  name: "Fido",
  speak() {
    console.log(this.name); // 'this' refers to 'dog'
  }
};
dog.speak(); // Fido
```

In contrast, in standalone functions or callbacks:

```
function speak() {
  console.log(this);
}
speak(); // 'this' is the global object (or undefined in strict mode)
```

Arrow functions and `this`:

Arrow functions do **not** create their own `this`. They **inherit it** from the surrounding (lexical) context.

Example:

```
function Timer() {
  this.seconds = 0;
```

```
  setInterval(() => {
    this.seconds++;
  }, 1000);
}
```

In this example, the arrow function inside `setInterval` uses the same `this` that was active when `Timer` was called. So `this.seconds++` works correctly.

What is the Timer object?

When you write:

```
const myTimer = new Timer();
```

JavaScript creates a new object, and inside the `Timer` function, `this` refers to that new object. The function adds a `seconds` property to that object and sets up a timer that increments it. So `myTimer.seconds` will increase every second.

Compare to a regular function (problematic):

```
function Timer() {
  this.seconds = 0;
  setInterval(function() {
    this.seconds++; // WRONG: 'this' does not refer to the Timer object
  }, 1000);
}
```

To fix this with a regular function, you'd often use `.bind(this)` or store `this` in a variable:

```
function Timer() {
  this.seconds = 0;
  const self = this;
  setInterval(function() {
    self.seconds++;
  }, 1000);
}
```

Arrow functions simplify this pattern.

Exercise 2: Rewrite this function using arrow syntax:

```
function greet(name) {
  return "Hello, " + name;
}
```

Answer:

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

Multiple-Choice:

What is one key difference between arrow functions and regular functions? A. Arrow functions can only return strings. B. Arrow functions require `bind` to access `this`. C. Arrow functions inherit `this` from the surrounding scope. D. Arrow functions are deprecated.

(Answer: C. Arrow functions inherit `this` from the surrounding scope.)

III. Static vs. Instance Methods in Classes (10 min)

- **Instance methods** operate on individual object data.
- **Static methods** belong to the class itself and are used for utility functions.

Example:

```
class MathTools {
  static double(x) {
    return x * 2;
  }

  square(x) {
    return x * x;
  }
}

console.log(MathTools.double(4)); // 8
const tool = new MathTools();
console.log(tool.square(3)); // 9
```

Exercise 3: Identify which of these should be static:

A. A method to calculate the average of two numbers. B. A method that logs the object's internal state.

(Answer: A = static; B = instance)

IV. Modern Syntax Patterns (15 min)

- Use default parameters:

```
function greet(name = "Guest") {
  return `Hello, ${name}`;
}
```

- Use concise object properties:

  If the variable name and the object property name are the same, you can omit the property name and just write the variable.

  This:

  ```
  const x = 10;
  const obj = { x }; // shorthand for { x: x }
  console.log(obj); // { x: 10 }
  ```

  is the same as:

  ```
  const x = 10;
  const obj = { x: x };
  ```

  It's a cleaner way to define objects when the variable name matches the key name.

  Discussion:

  While this shorthand is valid and widely used, it can break the mental model many programmers have: "Objects are key-value pairs written as `key: value`."

  In this shorthand, the key is implied — which can be confusing:

  ```
  const x = 10;
  const obj = { x }; // Looks like magic if you're expecting key: value
  format
  ```

  This can make code less readable or predictable. It's perfectly acceptable to use the full form:

  ```
  const obj = { x: x }; // clear, explicit, familiar
  ```

  Many developers (and some style guides) prefer the longer form when clarity is more important than brevity. Remember: **clarity is more important than cleverness**.

```
const x = 10;
const obj = { x };
console.log(obj); // { x: 10 }
```

- Use method shorthand in objects:

When defining methods inside object literals, you can omit the `function` keyword.

Example:

```
const math = {
  add(a, b) {
    return a + b;
  }
};
```

This is shorthand for:

```
const math = {
  add: function(a, b) {
    return a + b;
  }
};
```

The behavior is the same — it's just a more concise way to define methods.

Why use it?

- It saves space.
- It matches class method syntax.
- It's especially useful when creating collections of utility functions.

When not to use it:

- If you're returning an arrow function.
- If you want to clearly distinguish between properties and methods.

Some developers prefer always writing `function` explicitly to preserve clarity — both forms are valid.

Developer Note:

While method shorthand reduces typing, it comes with tradeoffs:

- It breaks the familiar `key: value` structure of object literals.
- It complicates grammar parsing and static analysis tools.
- It can confuse beginners who expect to see the `function` keyword.

This shorthand was introduced in ES6 to align object syntax with class method syntax, but many developers — especially those with experience in language design or formal programming models — find that the loss in clarity outweighs the minor gain in brevity.

Use with care. Clarity and consistency are often more valuable than saving keystrokes.

```
const math = {
  add(a, b) {
    return a + b;
  }
};
```

Exercise 4: Convert this to modern syntax:

```
function makePerson(name, age) {
  return {
    name: name,
    age: age,
    greet: function() {
      return "Hi, I'm " + name;
    }
  };
}
```

Answer:

```
function makePerson(name, age) {
  return {
    name,
    age,
    greet() {
      return `Hi, I'm ${name}`;
    }
  };
}
```

V. Recap & Q&A (10 min)

- Review template literals, arrow functions, `this` behavior, and method distinctions.
- Reinforce why arrow functions are concise and useful for handling lexical `this`.

Final Multiple-Choice Question:

Which statement is TRUE about arrow functions? A. They have their own `this`. B. They are always faster than regular functions. C. They inherit `this` from their surrounding context. D. They must always include curly braces.

(Answer: C. They inherit `this` from their surrounding context.)