guide # javascript

JavaScript for Java Developers

TCSS 460 — Client/Server Programming

Welcome to JavaScript. You already know how to program — you have been writing Java since TCSS 142, building data structures, designing OOP applications, and working on team projects. This guide is your bridge from Java to JavaScript. It will not teach you to program from scratch. Instead, it maps what you already know onto a new language so you can start building web applications as quickly as possible.

---

1 Why JavaScript?

JavaScript is the language of the web. Every browser on every device runs JavaScript natively, and with Node.js, it runs on servers too. That makes it the only language that works on both sides of a web application — the client that users interact with and the server that processes their requests.

In this course, you are building a full-stack web application: a PostgreSQL database, a Node.js/Express web API, and a Next.js front end. JavaScript (and its typed superset, TypeScript) is the common language across all three layers.

1.1 How Does JavaScript Compare to Java?

The names are similar, but that is mostly a historical marketing decision. Both are C-family languages, so the syntax will feel familiar — curly braces, semicolons, `if/else`, `for` loops. The behavior underneath, however, is quite different.

Feature	Java	JavaScript
Typing	Static (compiler checks types)	Dynamic (types checked at runtime)
Compilation	`javac` compiles to bytecode	No compilation step — interpreted

Feature	Java	JavaScript
Entry point	`public static void main(String[] args)`	Just start writing code
Classes	Required for everything	Optional — objects without classes
Running code	`javac File.java && java File`	`node file.js`

1.2 Where Does TypeScript Fit In?

TypeScript is JavaScript with a type system added on top. You will write TypeScript for every file in this course, but TypeScript compiles down to plain JavaScript before it runs. That means you need to understand JavaScript first — TypeScript builds on it, not replaces it.

Think of it this way: JavaScript is the language that actually runs. TypeScript is the safety net that catches mistakes before your code runs. This guide covers the JavaScript foundation. The next guide, *TypeScript Essentials*, adds the type system.

---

2 Running JavaScript

In Java, running a program requires multiple steps: write a `.java` file, compile it with `javac`, then run the bytecode with `java`. You also need a `public static void main` method as an entry point.

JavaScript has none of that. There is no compiler, no required entry point, and no class wrapper. You write code in a `.js` file and run it directly.

2.1 Running with Node.js

Node.js is a JavaScript runtime that lets you run JavaScript outside of a browser. It is already installed for this course (see the Node.js Setup guide).

Create a file called `hello.js`:

```
console.log("Hello from JavaScript!");
```

Run it from your terminal:

```
node hello.js
```

Output:

```
Hello from JavaScript!
```

That is it. No compilation step, no main method, no class. The file runs top to bottom.

2.2 The Node.js REPL

Node.js also includes a **REPL** (Read-Eval-Print Loop) — an interactive prompt where you can type JavaScript expressions and see results immediately. Start it by running `node` with no arguments:

```
node
```

You will see a `>` prompt. Type any expression:

```
> 2 + 3
5
> "Hello, " + "world!"
'Hello, world!'
> Math.random()
0.7291438520785432
```

Press `Ctrl+C` twice or type `.exit` to quit. The REPL is useful for quick experiments without creating a file.

Try It Yourself

1. Open your terminal and type `node`
2. Try some expressions: `2 + 3`, `"hello".toUpperCase()`, `[1, 2, 3].length`
3. Type `.exit` to quit

2.3 The Browser Console

Every web browser has a built-in JavaScript console. Open it with:

/ Windows / Linux

- **Chrome / Edge:** `^ Ctrl` + `⇧ Shift` + `J`
- **Firefox:** `^ Ctrl` + `⇧ Shift` + `K`

macOS

- **Chrome / Edge:** `⌘ Cmd` + `⌥ Option` + `J`
- **Firefox:** `⌘ Cmd` + `⌥ Option` + `K`

Type any JavaScript expression and press Enter to see the result immediately. This is great for quick experiments — no files needed.

> **Try It Yourself**
>
> 1. Open your browser's developer console
> 2. Type `2 + 3` and press Enter
> 3. Type `"Hello, " + "world!"` and press Enter
> 4. Type `Math.random()` and press Enter a few times

---

3 Variables: `let` and `const`

In Java, every variable needs a type declaration:

```
int count = 5;
String name = "Alice";
final double PI = 3.14159;
```

In JavaScript, you declare variables with `let` or `const` — no type annotation needed:

```
let count = 5;
const name = "Alice";
```

3.1 `let` — Reassignable Variables

Use `let` when the value will change:

```
let score = 0;
score = 10;       // allowed
```

```
score = score + 5; // allowed
console.log(score); // 15
```

3.2 `const` — Constants

Use `const` when the value should not be reassigned:

```
const maxRetries = 3;
maxRetries = 5; // TypeError: Assignment to constant variable
```

`const` is the default choice. Use `let` only when you know the variable needs to change.

> **Important**
>
> `const` prevents reassignment, not mutation. A `const` object can still have its properties changed — you just cannot point the variable at a different object. More on this in Section 8.

3.3 No `var`

You may see `var` in older JavaScript code or tutorials. Do not use it. `var` has confusing scoping rules that cause subtle bugs. Always use `let` or `const`.

3.4 No Type Declarations

Notice that JavaScript does not require you to write `int`, `String`, or any type name. JavaScript figures out the type at runtime based on the value you assign. This is called *dynamic typing*.

In Java, you write:

```
String greeting = "Hello";
int count = 42;
boolean active = true;
```

In JavaScript, the same thing is:

```
const greeting = "Hello";
let count = 42;
let active = true;
```

TypeScript will add type annotations back — but that is the next guide.

Try It Yourself

1. Create a file called `variables.js`
2. Add these lines:

```
const courseName = "TCSS 460";
let week = 1;
console.log("Welcome to " + courseName + ", Week " + week);
week = 2;
console.log("Now it's Week " + week);
```

3. Run it with `node variables.js`

---

4 Types in JavaScript (Dynamic Typing)

Java is statically typed — the compiler checks types before the program runs. JavaScript is dynamically typed — types are determined and checked at runtime.

4.1 The Primitive Types

JavaScript has fewer primitive types than Java:

JavaScript Type	Java Equivalent	Example
`string`	`String`	`"Hello"`
`number`	`int`, `double`, `float` (all combined)	`42`, `3.14`, `-7`
`boolean`	`boolean`	`true`, `false`
`null`	`null`	`null`
`undefined`	*(no equivalent)*	`undefined`

Notice that JavaScript has a single `number` type. There is no distinction between integers and floating-point numbers. `42` and `42.0` are the same type.

4.2 The `typeof` Operator

Since JavaScript does not declare types, you can check a value's type at runtime:

```
console.log(typeof "Hello");   // "string"
console.log(typeof 42);        // "number"
console.log(typeof true);      // "boolean"
console.log(typeof undefined); // "undefined"
console.log(typeof null);      // "object" (this is a famous JS bug)
```

⚠ The `typeof null` Bug

`typeof null` returns `"object"` instead of `"null"`. This is a well-known JavaScript bug that has existed since the language was created in 1995 and cannot be fixed without breaking the web. Just be aware of it.

4.3 `===` vs `==` (Always Use `===`)

JavaScript has two equality operators, and one of them is dangerous.

`===` **(strict equality)** — compares value AND type. This is what you want:

```
console.log(5 === 5);     // true
console.log(5 === "5");   // false (number vs string)
console.log(0 === false); // false (number vs boolean)
```

`==` **(loose equality)** — converts types before comparing. This leads to surprises:

```
console.log(5 == "5");     // true  (string "5" converted to number)
console.log(0 == false);   // true  (false converted to 0)
console.log("" == false);  // true  (both converted to 0)
console.log(null == undefined); // true
```

⚠ Always Use ===

The `==` operator's type coercion rules are complicated and produce unintuitive results. In this course — and in professional JavaScript — always use `===` for comparison and `!==` for inequality. Pretend `==` does not exist.

Try It Yourself

1. Open your browser console or create a `.js` file
2. Test these comparisons and predict the result before pressing Enter:

```
console.log(typeof 3.14);
console.log(typeof "3.14");
console.log(3.14 === "3.14");
console.log(3.14 == "3.14");
```

---

5 Strings

Strings in JavaScript are similar to Java's `String` class, with one major quality-of-life improvement: template literals.

5.1 Template Literals

In Java, building strings with variables is clunky:

```
String name = "Alice";
int age = 30;
String message = "Hello, " + name + "! You are " + age + " years old.";
// or with String.format:
String message = String.format("Hello, %s! You are %d years old.", name, age);
```

In JavaScript, use backticks (`` ` ``) with `${}` to embed expressions directly:

```
const name = "Alice";
const age = 30;
const message = `Hello, ${name}! You are ${age} years old.`;
console.log(message); // "Hello, Alice! You are 30 years old."
```

You can put any expression inside `${}`:

```
console.log(`2 + 3 = ${2 + 3}`);           // "2 + 3 = 5"
console.log(`Uppercase: ${name.toUpperCase()}`); // "Uppercase: ALICE"
```

Template literals also support multi-line strings without any escape characters:

```
const html = `
  <div>
    <h1>Hello, ${name}</h1>
    <p>Welcome to TCSS 460</p>
  </div>
`;
```

5.2 Useful String Methods

JavaScript strings have methods similar to Java's:

```
const text = "  Hello, World!  ";

console.log(text.includes("World"));    // true
console.log(text.startsWith("  Hello")); // true
console.log(text.trim());               // "Hello, World!"
console.log(text.split(","));           // ["  Hello", " World!  "]
console.log(text.toLowerCase());        // "  hello, world!  "
console.log(text.replace("World", "TCSS 460")); // "  Hello, TCSS 460!  "
```

JavaScript	Java Equivalent
`str.includes("x")`	`str.contains("x")`
`str.startsWith("x")`	`str.startsWith("x")`
`str.trim()`	`str.trim()`
`str.split(",")`	`str.split(",")`
`str.toLowerCase()`	`str.toLowerCase()`
`str.replace("a", "b")`	`str.replace("a", "b")`

---

6 Functions

Functions in JavaScript are more flexible than Java methods. In Java, every method lives inside a class. In JavaScript, functions are standalone and can be used in ways that Java does not allow.

6.1 Declaring a Function

In Java, you write a method with a return type, parameter types, and it must be inside a class:

```
public class Greeter {
    public static String greet(String name) {
        return "Hello, " + name;
    }
}
```

In JavaScript, a function is simpler — no class, no types:

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

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

No `public`, no `static`, no `String` return type, no `String` parameter type. Just the function name, parameters, and body.

6.2 Functions Are Values

This is the biggest conceptual leap from Java. In JavaScript, a function is a value — just like a number or a string. You can store it in a variable, pass it to another function, or return it from a function.

```
// Store a function in a variable
const greet = function(name) {
    return "Hello, " + name;
};

// Pass a function to another function
function runTwice(fn) {
    fn();
    fn();
}

runTwice(function() {
    console.log("Running!");
});
// Output:
// Running!
// Running!
```

In Java, you encountered this concept with lambdas (TCSS 305). In JavaScript, *all* functions work this way, not just special lambda expressions.

6.3 No Overloading

Java lets you define multiple methods with the same name but different parameter lists. JavaScript does not support overloading — each function name refers to exactly one function. Extra arguments are silently ignored, and missing arguments become `undefined`:

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

console.log(add(2, 3));       // 5
console.log(add(2, 3, 100));  // 5 (extra argument ignored)
console.log(add(2));          // NaN (b is undefined, 2 + undefined = NaN)
```

> **Missing Arguments Don't Throw Errors**
>
> In Java, calling a method with the wrong number of arguments is a compile error. In JavaScript, it silently proceeds with `undefined` for missing parameters. This can lead to confusing `NaN` or `undefined` results instead of helpful error messages. TypeScript fixes this by checking argument counts at compile time.

---

7 Arrow Functions

Arrow functions are a concise syntax for writing functions. You will see them everywhere in this course — especially in Express route handlers.

7.1 Basic Syntax

In Java, you write a lambda like this:

```
// Java lambda
Function<String, String> greet = (name) -> "Hello, " + name;
```

In JavaScript, the arrow function looks almost identical — just `=>` instead of `->`:

```
// JavaScript arrow function
const greet = (name) => "Hello, " + name;

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

7.2 Variations

Arrow functions have a few forms depending on the number of parameters and the complexity of the body:

```
// One parameter — parentheses optional
const double = n => n * 2;

// Multiple parameters — parentheses required
const add = (a, b) => a + b;

// No parameters — empty parentheses required
const sayHello = () => "Hello!";

// Multi-line body — use curly braces and explicit return
const greetFormal = (name) => {
    const greeting = `Good morning, ${name}`;
    return greeting + ". Welcome to TCSS 460.";
};
```

⚠ Implicit vs Explicit Return

Without curly braces, the expression after `=>` is automatically returned. With curly braces, you must use the `return` keyword explicitly. Forgetting `return` inside curly braces is a common bug:

```
// Returns the sum
const add = (a, b) => a + b;

// Returns undefined — missing return!
const addBroken = (a, b) => { a + b; };

// Fixed — explicit return
const addFixed = (a, b) => { return a + b; };
```

7.3 Why Arrow Functions Matter for This Course

In Express, every route handler is a function. You will write dozens of them. Arrow functions keep them concise:

```
// You'll write handlers like this all quarter
app.get("/hello", (req, res) => {
    res.json({ message: "Hello, World!" });
});

app.get("/users/:id", (req, res) => {
```

```
    const userId = req.params.id;
    res.json({ id: userId, name: "Alice" });
});
```

These are arrow functions receiving two parameters (`req` for the request, `res` for the response) and doing something with them. You will understand the full pattern in the Express guide — for now, just recognize the `(params) => { body }` syntax.

Try It Yourself

1. Create a file called `functions.js`
2. Write and test both styles:

```
// Traditional function
function square(n) {
    return n * n;
}

// Arrow function
const cube = (n) => n * n * n;

console.log(square(4));  // 16
console.log(cube(3));    // 27

// A function that takes another function as an argument
function applyToFive(fn) {
    return fn(5);
}

console.log(applyToFive(square)); // 25
console.log(applyToFive(cube));   // 125
console.log(applyToFive(n => n + 10)); // 15
```

3. Run it with `node functions.js`

---

8 Objects

In Java, creating a structured piece of data requires defining a class, writing a constructor, and (often) adding getters and setters. In JavaScript, you can create an object directly with an object literal — no class needed. This section covers the basics — for destructuring, spread syntax, and TypeScript interfaces, see the Objects, Arrays & Destructuring guide.

8.1 Object Literals

In Java, representing a user requires a class:

```
public class User {
    private String name;
    private int age;

    public User(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() { return name; }
    public int getAge() { return age; }
}

User alice = new User("Alice", 30);
```

In JavaScript, you just describe the shape:

```
const alice = { name: "Alice", age: 30 };
```

That is it. No class definition, no constructor, no getters. The object exists with two properties: `name` and `age`.

8.2 Accessing Properties

JavaScript objects support two ways to access properties:

```
const user = { name: "Alice", age: 30, email: "alice@uw.edu" };

// Dot notation (most common)
console.log(user.name);  // "Alice"
console.log(user.age);   // 30

// Bracket notation (useful when the key is dynamic)
const field = "email";
console.log(user[field]); // "alice@uw.edu"
```

In Java, this is roughly equivalent to a `HashMap<String, Object>` with dot-notation convenience, or a POJO without the class boilerplate.

8.3 Adding and Modifying Properties

Unlike Java objects, JavaScript objects are open — you can add or change properties at any time:

```
const user = { name: "Alice" };
user.age = 30;          // add a new property
user.name = "Bob";      // change an existing property
console.log(user);      // { name: "Bob", age: 30 }
```

Important

Notice that `user` was declared with `const`, yet we modified its properties. Remember: `const` prevents reassigning the variable to a different object. It does not prevent changing the object's contents. This is the same as Java's `final` — a `final` reference to an `ArrayList` still lets you add items to the list.

8.4 Nested Objects

Objects can contain other objects:

```
const student = {
    name: "Alice",
    university: {
        name: "UW Tacoma",
        department: "SET"
    },
    courses: ["TCSS 460", "TCSS 305"]
};

console.log(student.university.name); // "UW Tacoma"
console.log(student.courses[0]);      // "TCSS 460"
```

8.5 Why Objects Matter for This Course

Web APIs send and receive data as JSON (JavaScript Object Notation). JSON looks exactly like a JavaScript object literal. When you build an Express API, you will create, receive, and transform objects constantly:

```
// Sending a JSON response from an Express route
app.get("/user", (request, response) => {
    const user = { id: 1, name: "Alice", role: "student" };
    response.json(user);
});
```

The client receives:

```
{ "id": 1, "name": "Alice", "role": "student" }
```

Try It Yourself

1. Create a file called `objects.js`
2. Experiment with creating and accessing objects:

```
const course = {
    number: "TCSS 460",
    title: "Client/Server Programming",
    credits: 5,
    instructor: { name: "Charles Bryan", department: "SET" }
};

console.log(course.title);
console.log(course.instructor.name);

// Add a new property
course.quarter = "Spring 2026";
console.log(course);
```

3. Run it with `node objects.js`

---

9 Arrays

JavaScript arrays are similar to Java's `ArrayList` — they are dynamic (grow and shrink), zero-indexed, and have built-in methods for adding and removing elements. This section covers the basics — for `map`, `filter`, `find`, and other functional methods you will use constantly in Express routes, see the Array Methods guide.

9.1 Creating Arrays

In Java:

```
List<String> courses = new ArrayList<>();
courses.add("TCSS 460");
courses.add("TCSS 305");
// or in newer Java:
List<String> courses = List.of("TCSS 460", "TCSS 305");
```

In JavaScript:

```
const courses = ["TCSS 460", "TCSS 305"];
```

No type parameter, no `new`, no class name. Just square brackets.

9.2 Accessing and Modifying

```
const items = ["apple", "banana", "cherry"];

console.log(items[0]);     // "apple"
console.log(items.length); // 3

items.push("date");        // add to end
console.log(items);        // ["apple", "banana", "cherry", "date"]

items.pop();               // remove from end
console.log(items);        // ["apple", "banana", "cherry"]

console.log(items.includes("banana")); // true
```

JavaScript	Java Equivalent
`arr[0]`	`list.get(0)`
`arr.length`	`list.size()`
`arr.push(x)`	`list.add(x)`
`arr.pop()`	`list.remove(list.size()-1)`
`arr.includes(x)`	`list.contains(x)`

9.3 Mixed Types

Unlike Java's generic collections, JavaScript arrays can hold mixed types:

```
const mixed = [1, "two", true, null, { name: "Alice" }];
```

This is valid JavaScript. TypeScript will restrict this (which is a good thing), but in plain JavaScript, arrays have no type constraints.

Try It Yourself

1. Create a file called `arrays.js`
2. Try array operations:

```
const numbers = [10, 20, 30, 40, 50];

console.log("First:", numbers[0]);
console.log("Last:", numbers[numbers.length - 1]);

numbers.push(60);
console.log("After push:", numbers);

console.log("Has 30?", numbers.includes(30));
console.log("Has 99?", numbers.includes(99));
```

3. Run it with `node arrays.js`

---

10 Truthy and Falsy (No Java Equivalent)

This is one of the biggest differences from Java. In Java, `if` statements require a `boolean` expression. In JavaScript, *any* value can be used in a boolean context, and JavaScript will convert it to `true` or `false` automatically.

10.1 The Falsy Values

These six values are **falsy** — they evaluate to `false` in a boolean context:

Falsy Value	What It Is
`false`	The boolean false
`0`	The number zero
`""`	An empty string
`null`	Explicitly no value

Falsy Value	What It Is
`undefined`	Not assigned
`NaN`	"Not a Number" (invalid math result)

Everything else is truthy — including empty arrays `[]`, empty objects `{}`, and the string `"0"`.

10.2 Practical Usage

This pattern is extremely common in JavaScript:

```
const user = getUser(); // might return a user object or undefined

if (user) {
    console.log(`Welcome, ${user.name}`);
} else {
    console.log("User not found");
}
```

Instead of Java's explicit `if (user != null)`, JavaScript developers write `if (user)`. This works because `undefined` and `null` are falsy, while any object is truthy.

10.3 The Gotchas

⚠ Watch Out for `0` and Empty Strings

Because `0` and `""` are falsy, this pattern can produce bugs:

```
const count = 0;  // valid count — zero items

if (count) {
    console.log(`Found ${count} items`);
} else {
    console.log("No count provided"); // This runs! 0 is falsy.
}
```

If zero is a valid value, check explicitly:

```
if (count !== undefined && count !== null) {
    console.log(`Found ${count} items`); // Now this runs for 0
}
```

---

11 Control Flow

Good news — control flow in JavaScript is almost identical to Java.

11.1 `if/else`

```
const score = 85;

if (score >= 90) {
    console.log("A");
} else if (score >= 80) {
    console.log("B");
} else {
    console.log("Below B");
}
```

Exactly the same syntax as Java.

11.2 `for` Loops

The classic `for` loop works identically:

```
for (let i = 0; i < 5; i++) {
    console.log(i);
}
```

11.3 `for...of` — Iterating Over Values

JavaScript has a cleaner way to iterate over arrays:

In Java:

```
List<String> names = List.of("Alice", "Bob", "Charlie");
for (String name : names) {
    System.out.println(name);
}
```

In JavaScript:

```
const names = ["Alice", "Bob", "Charlie"];
for (const name of names) {
    console.log(name);
}
```

The `for...of` loop iterates directly over values. No index variable, no `.get()` calls, no iterator boilerplate.

11.4 `while` Loops

Also identical to Java:

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

---

12 Logical Operators as Expressions

In Java, `&&` and `||` always return a `boolean`. JavaScript works differently — logical operators return one of the operand values, not `true` or `false`. This is a direct consequence of the truthy/falsy system from Section 10, and it unlocks patterns you will use constantly in Express.

12.1 `||` — First Truthy Value

The `||` operator evaluates left to right and returns the **first truthy value** it finds. If all values are falsy, it returns the last value:

```
console.log("" || "default");         // "default" (empty string is falsy)
console.log("hello" || "default");    // "hello" (already truthy — stops here)
console.log(0 || "" || "fallback");   // "fallback" (0 and "" are both falsy)
console.log(0 || "" || null);         // null (all falsy — returns the last
one)
```

Compare this to Java, where `"" || "default"` would not even compile — Java's `||` only works with `boolean` operands.

12.2 `&&` — First Falsy Value

The `&&` operator evaluates left to right and returns the **first falsy value** it finds. If all values are truthy, it returns the last value:

```
console.log(null && "anything");      // null (null is falsy — stops here)
console.log("hello" && "world");      // "world" (both truthy — returns last)
console.log(1 && 2 && 3);            // 3 (all truthy — returns last)
console.log(1 && 0 && 3);            // 0 (first falsy value)
```

A common use is safe property access:

```
const user = null;
console.log(user && user.name);       // null (doesn't try to access .name)

const admin = { name: "Alice" };
console.log(admin && admin.name);     // "Alice"
```

If `user` is `null` (falsy), `&&` returns `null` immediately without evaluating `user.name` — avoiding the `TypeError` you would get from `null.name`.

12.3 `??` — Nullish Coalescing

The `??` operator is newer (ES2020) and more precise than `||`. It returns the right-hand side **only** when the left-hand side is `null` or `undefined` — not for other falsy values like `0` or `""`:

```
console.log(null ?? "default");       // "default"
console.log(undefined ?? "default"); // "default"
console.log(0 ?? "default");         // 0 (0 is NOT null/undefined)
console.log("" ?? "default");        // "" (empty string is NOT null/undefined)
console.log(false ?? "default");     // false (false is NOT null/undefined)
```

This distinction matters more than you might expect.

12.4 The `??` vs `||` Trap

⚠ The ?? vs || Trap

Using `||` for default values silently replaces `0`, `""`, and `false` — which might be perfectly valid values:

```
const count = 0;
console.log(count || 10);   // 10 — WRONG! 0 was a valid count
console.log(count ?? 10);   // 0 — correct! 0 is not null/undefined

const title = "";
console.log(title || "Untitled"); // "Untitled" — WRONG if "" was intentional
console.log(title ?? "Untitled"); // "" — correct!
```

Rule of thumb: Use `??` when you want a fallback for missing values. Use `||` only when you genuinely want to replace all falsy values.

Here is the comparison at a glance:

Left Value	`left || right`	`left ?? right`	Why Different?
`null`	Returns right	Returns right	Both treat `null` as "use fallback"
`undefined`	Returns right	Returns right	Both treat `undefined` as "use fallback"
`0`	Returns right	Returns **left**	`||` sees `0` as falsy; `??` keeps it
`""`	Returns right	Returns **left**	`||` sees `""` as falsy; `??` keeps it
`false`	Returns right	Returns **left**	`||` sees `false` as falsy; `??` keeps it

12.5 Default Values in Express

These operators show up constantly when handling request parameters in Express. Query parameters from a URL are always strings or `undefined` — you need sensible defaults:

```
// GET /users?page=2&limit=50

app.get("/users", (request, response) => {
    // ?? is the right choice — limit=0 should mean 0, not 10
    const limit = request.query.limit ?? 10;
    const page = request.query.page ?? 1;

    // || would be wrong here — if someone sends ?limit=0,
    // you'd silently replace it with 10
});
```

You will write patterns like this in every Express route that accepts optional parameters.

Try It Yourself

1. Open your browser console or Node.js REPL
2. Test these expressions and predict each result before pressing Enter:

```
console.log("" || "fallback");
console.log("" ?? "fallback");
console.log(0 || 42);
console.log(0 ?? 42);
console.log(null || "backup");
console.log(null ?? "backup");
console.log(undefined ?? "yes");
console.log(false || true);
console.log(false ?? true);
```

---

13 `null` vs `undefined`

Java has one way to represent "no value": `null`. JavaScript has two: `null` and `undefined`. This catches every Java developer off guard.

13.1 `undefined` — "Not Yet Assigned"

A variable that has been declared but not assigned a value is `undefined`. A missing object property is also `undefined`:

```
let x;
console.log(x); // undefined

const user = { name: "Alice" };
console.log(user.age); // undefined (property doesn't exist)
```

13.2 `null` — "Explicitly Nothing"

`null` is an intentional assignment meaning "this has no value":

```
let selectedUser = null; // explicitly no user selected
```

13.3 How to Check for Both

In practice, you often want to check for either `null` or `undefined`. A common pattern:

```
// Check for null or undefined (both are falsy)
if (value == null) {
    // value is null OR undefined
}
```

This is the one legitimate use of `==` (loose equality) — `null == undefined` is `true`, and no other values match. That said, many teams still prefer the explicit check:

```
if (value === null || value === undefined) {
    // explicit and clear
}
```

Gen AI & Learning: Handling Null Values

When working with AI coding assistants, you will notice they frequently generate null-checking patterns. Understanding the difference between `null` and `undefined` helps you evaluate whether generated code handles edge cases correctly. If an AI suggests `if (value)` but `0` or `""` are valid inputs, you will know to change it to `if (value !== null && value !== undefined)`. The ability to spot these subtleties is what separates a developer who uses AI effectively from one who copies blindly.

14 Console and Debugging

In Java, you use `System.out.println()` for output. JavaScript's equivalent is `console.log()`.

14.1 console.log()

Your primary debugging tool:

```
const name = "Alice";
const age = 30;

console.log("Name:", name);          // Name: Alice
console.log("Age:", age);            // Age: 30
console.log({ name, age });          // { name: 'Alice', age: 30 }
```

> **Tip**
>
> Passing an object to `console.log()` prints its structure, which is much more readable than Java's default `toString()` that prints a memory address.

14.2 Other Console Methods

```
// Error output (shows in red in most consoles)
console.error("Something went wrong!");

// Tabular data (great for arrays of objects)
const users = [
    { name: "Alice", age: 30 },
    { name: "Bob", age: 25 }
];
console.table(users);

// Timing
console.time("operation");
// ... some code ...
console.timeEnd("operation"); // operation: 12.345ms
```

`console.table()` is particularly useful when working with API responses — it formats arrays of objects as a readable table in your terminal.

Try It Yourself

1. Create a file called `console-demo.js`
2. Try all the console methods:

```
console.log("Regular log");
console.error("This is an error");

const students = [
    { name: "Alice", grade: "A" },
    { name: "Bob", grade: "B" },
    { name: "Charlie", grade: "A" }
];
console.table(students);

console.time("loop");
for (let i = 0; i < 1000000; i++) { /* nothing */ }
console.timeEnd("loop");
```

3. Run it with `node console-demo.js`

---

15 Summary

You have covered a lot of ground. Here is what you now know — and how it maps to what you already knew from Java:

Concept	Java	JavaScript
Variable declaration	`int x = 5;`	`let x = 5;` or `const x = 5;`
Type system	Static (compile-time)	Dynamic (runtime)
Strings	`"Hello" + name` or `String.format(...)`	`` `Hello, ${name}` ``
Functions	Methods inside classes	Standalone, first-class values

Concept	Java	JavaScript
Arrow functions	`(x) -> x * 2`	`(x) => x * 2`
Objects	Class with constructor	`{ name: "Alice", age: 30 }`
Arrays	`ArrayList<String>`	`["a", "b", "c"]`
For-each loop	`for (String s : list)`	`for (const s of list)`
Null handling	`null` only	`null` and `undefined`
Boolean coercion	Not allowed	Everything is truthy or falsy
Equality	`==` and `.equals()`	`===` (always use strict)
Output	`System.out.println()`	`console.log()`
Running code	`javac` then `java`	`node file.js`

The syntax is familiar. The concepts underneath — dynamic typing, functions as values, truthy/falsy, two kinds of null — are what you need to internalize. Do not worry if it feels odd at first. You have been writing Java for years, and this is genuinely different in important ways. Give yourself time to adjust.

Next up: the **TypeScript Essentials** guide adds the type system back on top of everything you learned here — giving you the safety net of static types with the flexibility of JavaScript.

---

16 References

Official Documentation:

- MDN JavaScript Guide — Mozilla's comprehensive JavaScript reference. The "Introduction" and "Grammar and types" sections map well to this guide.
- MDN JavaScript Reference — Detailed reference for every built-in object, statement, and operator.

- Node.js Documentation — Official API docs for Node.js, the runtime you will use throughout this course.

Tutorials:

- JavaScript.info — The Modern JavaScript Tutorial — Well-structured tutorial covering JavaScript fundamentals through advanced topics. Chapters 1-5 align with this guide.

---

17 Further Reading

External Resources

- MDN: JavaScript First Steps — Mozilla's beginner learning path for JavaScript
- MDN: Equality Comparisons and Sameness — Deep dive into `==` vs `===` and the type coercion rules
- MDN: Falsy Values — Complete list and explanation of falsy values
- Node.js: Introduction to Node.js — Official getting-started guide for the Node.js runtime

---

This guide is part of TCSS 460 — Client/Server Programming, School of Engineering and Technology, University of Washington Tacoma.