guide # typescript

Objects, Arrays & Destructuring

TCSS 460 — Client/Server Programming

Objects and arrays are the fundamental data structures you will work with everywhere in this course — Express route handlers, database query results, API responses, and React component props. Destructuring — the ability to unpack values from objects and arrays into distinct variables — is not optional syntax sugar in TypeScript. It is the standard way to write modern TypeScript code. By the end of this guide, patterns like `const { id } = request.params` will feel as natural as calling a getter in Java.

---

1 Objects in Depth

If you have read the *JavaScript for Java Developers* guide, you have already seen object literals. This section goes deeper into how objects work, how to access and modify them, and the patterns you will use constantly in Express.

1.1 Creating Objects

In Java, creating a structured piece of data typically requires defining 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 user = new User("Alice", 30);
```

In TypeScript, you define an interface for the shape and create an object literal directly:

```
interface User {
    name: string;
    age: number;
}

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

No constructor, no getters, no boilerplate. The interface describes the shape, and the object literal satisfies it. This is **structural typing** — if the object has the right properties with the right types, it fits the interface.

You can also create objects without an explicit interface when the type can be inferred:

```
const config = {
    port: 3000,
    host: "localhost",
    debug: true,
};
// TypeScript infers: { port: number; host: string; debug: boolean }
```

1.2 Accessing Properties: Dot vs. Bracket Notation

There are two ways to read a property from an object:

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

// Dot notation — use this most of the time
console.log(user.name);  // "Alice"

// Bracket notation — use when the key is dynamic or not a valid identifier
console.log(user["name"]);  // "Alice"
```

Dot notation is cleaner and what you will use in the vast majority of cases. Bracket notation becomes necessary when:

- The property name is stored in a variable
- The property name contains special characters (hyphens, spaces)
- You are iterating over keys dynamically

```
const field = "name";
console.log(user[field]);  // "Alice" — dot notation can't do this

// HTTP headers often have hyphens
const headers: Record<string, string> = {
    "content-type": "application/json",
    "x-request-id": "abc123",
};
console.log(headers["content-type"]);  // "application/json"
```

1.3 Modifying Properties

If an object is declared with `const`, you cannot reassign the variable, but you **can** modify its properties:

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

// This works — modifying a property
user.age = 31;

// This fails — reassigning the variable
// user = { name: "Bob", age: 25 };  // Error: Cannot assign to 'user'
```

This catches Java developers off guard. In Java, `final` on a reference variable prevents reassignment, and the fields themselves must be `final` to be truly immutable. TypeScript works the same way — `const` only prevents reassignment of the variable, not mutation of the object.

To make an object truly immutable, use `Readonly<T>`:

```
const user: Readonly<User> = { name: "Alice", age: 30 };
// user.age = 31;  // Error: Cannot assign to 'age' because it is a read-only
property
```

1.4 Computed Property Names

You can use an expression inside brackets to create a property name dynamically:

```
const key = "email";

const user = {
    name: "Alice",
    [key]: "alice@example.com",  // equivalent to: email: "alice@example.com"
};

console.log(user.email);  // "alice@example.com"
```

This is useful when building objects programmatically, such as constructing query parameters or transforming API responses where the field names come from variables or configuration.

---

2 Spread Operator for Objects

The spread operator (`...`) is one of the most useful features in TypeScript, and it has no clean equivalent in Java. It lets you copy all properties from one object into a new one.

2.1 Copying Objects

```
const original = { name: "Alice", age: 30, email: "alice@example.com" };

// Create a shallow copy
const copy = { ...original };

console.log(copy);  // { name: "Alice", age: 30, email: "alice@example.com" }
console.log(copy === original);  // false — it's a new object
```

In Java, you would need to write a copy constructor, implement `Cloneable`, or manually copy each field. The spread operator does this in one expression.

2.2 Overriding Properties

The real power of spread is combining a copy with overrides. Properties listed **after** the spread take precedence:

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

// Create a new object with the role changed
const admin = { ...user, role: "admin" };

console.log(admin);  // { name: "Alice", age: 30, role: "admin" }
console.log(user);   // { name: "Alice", age: 30, role: "user" } — unchanged
```

This pattern is everywhere in web development. Instead of mutating an existing object, you create a new one with the changes applied. It is especially common when:

- Building API response objects from database results
- Merging default configuration with user-provided options
- Updating state without mutation

```
// Merging defaults with user config
const defaults = { port: 3000, host: "localhost", debug: false };
const userConfig = { port: 8080, debug: true };

const config = { ...defaults, ...userConfig };
// { port: 8080, host: "localhost", debug: true }
```

2.3 Merging Multiple Objects

You can spread multiple objects into one. Later properties overwrite earlier ones:

```
const base = { a: 1, b: 2 };
const override = { b: 3, c: 4 };
const extra = { d: 5 };

const merged = { ...base, ...override, ...extra };
// { a: 1, b: 3, c: 4, d: 5 }
```

> ⚠ **Shallow Copy Only**
>
> The spread operator creates a **shallow** copy. If a property's value is an object or array, the copy shares a reference to that nested value — it does not deep-clone it.
>
> ```
> const original = {
>     name: "Alice",
>     address: { city: "Tacoma", state: "WA" },
> };
>
> const copy = { ...original };
> copy.address.city = "Seattle";
>
> console.log(original.address.city);  // "Seattle" — both point
> to the same address object
> ```
>
> For most Express work, shallow copies are sufficient. If you need a deep copy, use `structuredClone()`:
>
> ```
> const deepCopy = structuredClone(original);
> ```

---

3 Destructuring Objects

Destructuring is the syntax for extracting values from an object (or array) into individual variables. If spread is about putting things together, destructuring is about taking them apart.

3.1 Basic Object Destructuring

Instead of accessing properties one at a time:

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

// Without destructuring
const name = user.name;
const age = user.age;
const email = user.email;
```

You can extract all three in a single statement:

```
const { name, age, email } = user;

console.log(name);   // "Alice"
console.log(age);    // 30
console.log(email);  // "alice@example.com"
```

The variable names must match the property names. The curly braces on the left side of the assignment are **not** creating an object — they are destructuring one.

> **This Is Not Optional Syntax**
>
> In Java, you access fields through getters: `user.getName()`. In TypeScript, you will see destructuring in virtually every Express route handler, every middleware function, and every utility. It is the standard way to extract values. You need to be comfortable reading and writing it.

3.2 Renaming Variables

Sometimes the property name conflicts with an existing variable, or you want a more descriptive name. Use a colon to rename:

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

// Rename 'name' to 'userName'
const { name: userName, age: userAge } = user;

console.log(userName);  // "Alice"
console.log(userAge);   // 30
// console.log(name);   // Error — 'name' was not created, 'userName' was
```

> **The Colon Means Rename, Not Type**
>
> In destructuring, `{ name: userName }` means "take the `name` property and store it in a variable called `userName`." It does **not** mean "name has type userName." This is a common source of confusion because TypeScript also uses colons for type annotations.

3.3 Default Values

If a property might be `undefined`, you can provide a fallback:

```
interface Config {
    port?: number;
    host?: string;
    debug?: boolean;
}

const config: Config = { port: 8080 };

const { port, host = "localhost", debug = false } = config;

console.log(port);   // 8080
console.log(host);   // "localhost" — default applied
console.log(debug);  // false — default applied
```

Default values only kick in when the property is `undefined`, not when it is `null`. This matters in Express when query parameters may be missing entirely.

3.4 Extracting Only What You Need

You do not have to destructure every property. Extract only what you need:

```
const user = { name: "Alice", age: 30, email: "alice@example.com", role:
"admin" };

// Only extract name and role
const { name, role } = user;
```

This is a significant advantage over Java, where accessing multiple fields from an object requires multiple lines of getter calls regardless of how many you need.

Try It Yourself

1. Create a file called `destructure.ts`
2. Define an interface `Product` with `name`, `price`, and `category` properties
3. Create a product object and destructure it, renaming `name` to `productName`
4. Add a default value for an optional `discount` property
5. Run it with `npx ts-node destructure.ts`

---

4 Arrays in Depth

Arrays in TypeScript are typed and flexible. Unlike Java's `ArrayList` that requires generics and wrapper types, TypeScript arrays work directly with primitives and objects.

4.1 Creating and Typing Arrays

```
// Type annotation with bracket syntax
const numbers: number[] = [1, 2, 3, 4, 5];

// Generic syntax (equivalent)
const names: Array<string> = ["Alice", "Bob", "Charlie"];

// TypeScript infers the type when you initialize
const scores = [95, 87, 92];  // inferred as number[]
```

In Java, you choose between arrays (`int[]`) and `ArrayList<Integer>`. In TypeScript, `number[]` gives you the flexibility of `ArrayList` with the syntax of an array.

4.2 Common Array Methods

TypeScript arrays come with many built-in methods. Here are the ones you will use most often:

Adding and removing elements:

```
const items: string[] = ["a", "b", "c"];

items.push("d");       // Add to end → ["a", "b", "c", "d"]
items.pop();           // Remove from end → ["a", "b", "c"]
items.unshift("z");    // Add to start → ["z", "a", "b", "c"]
items.shift();         // Remove from start → ["a", "b", "c"]
```

Slicing (non-destructive) vs. splicing (destructive):

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

// slice — returns a new array, original unchanged
const middle = arr.slice(1, 4);  // [20, 30, 40]
console.log(arr);                // [10, 20, 30, 40, 50] — unchanged

// splice — modifies the original array
arr.splice(2, 1);         // Remove 1 element at index 2
console.log(arr);          // [10, 20, 40, 50]
```

```
arr.splice(1, 0, 15, 25); // Insert 15 and 25 at index 1
console.log(arr);          // [10, 15, 25, 20, 40, 50]
```

> ⚠ **`splice` Mutates, `slice` Does Not**
>
> This is one of the most common sources of bugs. `slice` creates a new array and leaves the original alone. `splice` modifies the original array in place. When in doubt, prefer `slice` and the spread operator for immutable patterns.

Searching:

```
const users = ["Alice", "Bob", "Charlie"];

users.includes("Bob");     // true
users.indexOf("Charlie");  // 2
users.indexOf("Dave");     // -1 (not found)
```

4.3 Spread Operator for Arrays

Just like objects, the spread operator works on arrays:

```
const frontend = ["React", "Next.js"];
const backend = ["Express", "Prisma"];

// Combine arrays
const fullStack = [...frontend, ...backend];
// ["React", "Next.js", "Express", "Prisma"]

// Copy an array
const copy = [...frontend];

// Add elements while spreading
const withNode = ["Node.js", ...backend, "PostgreSQL"];
// ["Node.js", "Express", "Prisma", "PostgreSQL"]
```

In Java, combining two `ArrayList` instances requires `addAll()` or stream operations. The spread operator is more concise and creates a new array rather than modifying an existing one.

Try It Yourself

1. Create two arrays of your favorite movies (3 each)
2. Combine them with spread into a single array
3. Use `slice` to get the middle two elements
4. Verify the original arrays are unchanged

---

5 Destructuring Arrays

Array destructuring extracts values by **position** rather than by name.

5.1 Basic Array Destructuring

```
const colors = ["red", "green", "blue"];

const [first, second, third] = colors;

console.log(first);   // "red"
console.log(second);  // "green"
console.log(third);   // "blue"
```

Notice the square brackets on the left side. This is how TypeScript distinguishes array destructuring from object destructuring — square brackets for arrays, curly braces for objects.

5.2 Skipping Elements

Use commas to skip positions you do not need:

```
const rgb = [255, 128, 0];

const [red, , blue] = rgb;  // Skip the second element

console.log(red);   // 255
console.log(blue);  // 0
```

5.3 The Rest Element

The rest element (`...`) collects remaining items into a new array:

```
const numbers = [1, 2, 3, 4, 5];

const [first, second, ...rest] = numbers;

console.log(first);   // 1
console.log(second);  // 2
console.log(rest);    // [3, 4, 5]
```

The rest element must be the last element in the destructuring pattern. This is useful for separating the "head" of a list from its "tail," a pattern familiar from data structures courses.

5.4 Default Values

Just like object destructuring, you can provide defaults:

```
const pair = [42];

const [x, y = 0] = pair;

console.log(x);  // 42
console.log(y);  // 0 — default applied because pair[1] is undefined
```

5.5 Swapping Variables

Array destructuring enables a clean swap without a temporary variable:

```
let a = 1;
let b = 2;

[a, b] = [b, a];

console.log(a);  // 2
console.log(b);  // 1
```

In Java, swapping two variables always requires a temporary variable (or XOR tricks). This is a small but satisfying improvement.

---

6 Nested Destructuring

Objects often contain other objects. You can destructure nested properties in a single expression.

6.1 Destructuring Nested Objects

```
interface Address {
    city: string;
    state: string;
    zip: string;
}

interface User {
    name: string;
    age: number;
    address: Address;
}

const user: User = {
    name: "Alice",
    age: 30,
    address: {
        city: "Tacoma",
        state: "WA",
        zip: "98402",
    },
};

// Nested destructuring — extract city directly
const { address: { city } } = user;

console.log(city);  // "Tacoma"
```

⚠ The Parent Variable Is Not Created

In the example above, `const { address: { city } } = user` creates a variable `city` but does **not** create a variable `address`. The `address:` part tells TypeScript to "go into the address property," not to create a variable called `address`.

If you need both the parent object and a nested value, destructure them separately:

```
const { address } = user;
const { city } = address;
```

6.2 When Nesting Gets Too Deep

Nested destructuring can become unreadable quickly:

```
// Too deep — hard to read, hard to debug
const { company: { headquarters: { address: { city } } } } = data;
```

When destructuring goes beyond two levels, break it into intermediate steps:

```
// Much clearer
const { company } = data;
const { headquarters } = company;
const { city } = headquarters.address;
```

> **Rule of Thumb**
>
> One level of nesting is fine: `const { address: { city } } = user`. Two levels is pushing it. Three or more — use intermediate variables. Your future self (and your teammates) will thank you.

6.3 Nested Arrays

You can combine array and object destructuring:

```
const response = {
    status: 200,
    data: {
        results: ["Inception", "Interstellar", "The Dark Knight"],
    },
};

const { data: { results: [firstMovie] } } = response;

console.log(firstMovie);  // "Inception"
```

While powerful, this kind of deeply nested destructuring is best reserved for cases where you only need one or two values. For complex API responses, consider extracting step by step.

---

7 Destructuring in Function Parameters

This is where destructuring goes from "nice syntax" to "essential skill." In Express, nearly every route handler destructures something from the request object.

7.1 Object Parameters

Instead of accepting a whole object and accessing its properties inside the function:

```
// Without destructuring
function greet(user: User): string {
    return `Hello, ${user.name}! You are ${user.age} years old.`;
}
```

You can destructure the parameter directly:

```
// With destructuring
function greet({ name, age }: User): string {
    return `Hello, ${name}! You are ${age} years old.`;
}
```

Both versions do the same thing. The destructured version is more concise and makes it immediately clear which properties the function uses.

7.2 Default Values in Parameters

Combine destructuring with default values for configuration-style functions:

```
interface ServerConfig {
    port?: number;
    host?: string;
    debug?: boolean;
}

function startServer({ port = 3000, host = "localhost", debug = false }:
ServerConfig): void {
    console.log(`Server running on ${host}:${port}`);
    if (debug) {
        console.log("Debug mode enabled");
    }
}

startServer({ port: 8080 });               // Server running on localhost:8080
startServer({ port: 8080, debug: true });   // Server running on
localhost:8080 + debug
startServer({});                            // Server running on
localhost:3000
```

In Java, you would typically handle this with method overloading or a builder pattern — far more verbose for the same result.

7.3 The Express Pattern

Here is why this section matters most. Every Express route handler receives `request` and `response` objects. The request object (`request`) carries data from the client in several places:

Property	Source	Example
`request.params`	URL path parameters	`/users/:id` gives `request.params.id`
`request.query`	Query string	`/users?role=admin` gives `request.query.role`
`request.body`	Request body (POST/PUT)	JSON payload gives `request.body.name`

Destructuring is the standard way to extract these values:

```
import express, { Request, Response } from "express";

const app = express();
app.use(express.json());

// GET /users/:id — extract id from URL params
app.get("/users/:id", (request: Request, response: Response) => {
    const { id } = request.params;
    // Use id to look up the user
    response.json({ userId: id });
});

// GET /users?role=admin&active=true — extract from query string
app.get("/users", (request: Request, response: Response) => {
    const { role, active } = request.query;
    // Use role and active to filter users
    response.json({ role, active });
});

// POST /users — extract from request body
app.post("/users", (request: Request, response: Response) => {
    const { name, email, age } = request.body;
    // Use name, email, age to create a user
    response.status(201).json({ name, email, age });
});
```

You Will See This in Every Route Handler

The pattern `const { id } = request.params` is not a stylistic choice — it is how Express code is written in practice. When you read the starter code for your check-offs and group project, you will see this on nearly every line that touches request data. Understanding destructuring is a prerequisite for understanding Express.

7.4 Combining Multiple Destructures

A single route handler often extracts from multiple sources:

```
// PUT /users/:id — update a user
app.put("/users/:id", (request: Request, response: Response) => {
    const { id } = request.params;              // Who to update
    const { name, email } = request.body;       // What to update

    // Now use id, name, email...
    response.json({ id, name, email });
});
```

Notice that the response also uses **shorthand property names**: `{ id, name, email }` instead of `{ id: id, name: name, email: email }`. When the variable name matches the property name, you can omit the repetition. This is another common TypeScript pattern you will see alongside destructuring.

7.5 Destructuring with Validation

In real Express handlers, you should validate the destructured values before using them:

```
app.post("/users", (request: Request, response: Response) => {
    const { name, email } = request.body;

    if (!name || !email) {
        response.status(400).json({ error: "Name and email are required" });
        return;
    }

    // Safe to use name and email here
    response.status(201).json({ name, email });
});
```

This pattern — destructure, validate, proceed — is the foundation of input handling in Express. You will build on it throughout the quarter.

Gen AI & Learning: Destructuring and Code Generation

When you use an AI coding agent to scaffold Express route handlers, it will write destructuring by default. Understanding what `const { id } = request.params` means — and being able to modify it when the generated code does not match your API design — is essential. The agent generates the pattern; you need to understand and adjust it.

Try It Yourself

1. Create a file called `express-practice.ts`
2. Write a function `handleCreatePost` that takes an Express-style `body` parameter with `title`, `content`, and optional `tags` (default to empty array)
3. Destructure all three properties in the function signature
4. Return a formatted string that includes the title and the number of tags
5. Test it by calling the function with `{ title: "Hello", content: "World" }` and `{ title: "Hello", content: "World", tags: ["ts", "express"] }`

---

8 Summary

Concept	Key Point
Object literals	Create structured data without classes — `const user = { name: "Alice" }`
Dot vs. bracket notation	Use dot notation by default; bracket notation when the key is dynamic
Spread operator (objects)	`{ ...original, key: "new" }` creates a shallow copy with overrides
Spread operator (arrays)	`[...arr1, ...arr2]` combines arrays into a new one
Object destructuring	`const { name, age } = user` extracts properties into variables
Renaming in destructuring	`const { name: userName } = user` renames the variable
Default values	`const { port = 3000 } = config` provides fallbacks for missing properties
Array destructuring	`const [first, ...rest] = items` extracts by position

Concept	Key Point
Nested destructuring	`const { address: { city } } = user` reaches into nested objects
Function parameter destructuring	`function greet({ name }: User)` extracts in the signature
Express pattern	`const { id } = request.params` is the standard way to access request data
Shorthand properties	`{ id, name }` is shorthand for `{ id: id, name: name }`
Shallow copy warning	Spread only copies one level deep — nested objects share references

---

9 References

Official Documentation:

- TypeScript Handbook — Object Types — Interfaces, optional properties, and structural typing
- MDN Web Docs — Destructuring Assignment — Complete reference for destructuring syntax
- MDN Web Docs — Spread Syntax — Spread for arrays and objects
- MDN Web Docs — Object Initializer — Object literals, computed properties, shorthand
- Express.js — req API — Request object properties (`params` , `query` , `body`)

---

10 Further Reading

External Resources

- JavaScript.info — Destructuring Assignment - Thorough walkthrough with interactive examples covering both object and array destructuring
- JavaScript.info — Object Spread and Rest - In-depth coverage of the spread and rest operators
- TypeScript Deep Dive — Spread Operator - TypeScript-specific perspective on spread with type implications

---

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