guide # typescript

TypeScript Essentials

TCSS 460 — Client/Server Programming

You just learned JavaScript — a dynamically typed language where variables can hold any value and errors hide until runtime. Now you are adding TypeScript on top, bringing back the compile-time type checking you relied on in Java. This guide covers type annotations, interfaces, unions, and the core patterns you will use in every `.ts` file for the rest of this course.

---

1 Why TypeScript?

JavaScript has a fundamental problem for building reliable software: it catches errors too late. Consider this JavaScript function:

```
function calculateTotal(price, quantity) {
    return price * quantity;
}

calculateTotal("ten", 3); // NaN — no error, just wrong
```

JavaScript happily multiplies the string `"ten"` by `3` and produces `NaN` (Not a Number). No error is thrown. No warning appears. Your API returns garbage data, and you discover the problem when a user files a bug report — or worse, when you are debugging at 2 AM.

1.1 The Java Analogy

You already know why type checking matters. In Java, this would never compile:

```
public int calculateTotal(int price, int quantity) {
    return price * quantity;
}

calculateTotal("ten", 3); // Compiler error: incompatible types
```

The Java compiler stops you immediately. TypeScript brings that same protection to the JavaScript ecosystem:

```
function calculateTotal(price: number, quantity: number): number {
    return price * quantity;
}

calculateTotal("ten", 3); // TypeScript error: Argument of type 'string'
                          // is not assignable to parameter of type 'number'
```

TypeScript catches the bug before your code ever runs. The error appears in your editor as you type, with a red squiggly underline and a clear message explaining what went wrong.

1.2 TypeScript in This Course

Every file you write in TCSS 460 is a `.ts` file, not `.js`. The TypeScript compiler (`tsc`) checks your types and then produces plain JavaScript that Node.js can run. You get the safety of static types during development and the runtime compatibility of JavaScript in production.

Here is the mental model:

Phase	What Happens	Analogy
You write	`.ts` files with type annotations	Writing `.java` files
TypeScript checks	Types verified, errors reported	`javac` compilation
TypeScript emits	Plain `.js` files	`.class` bytecode
Node.js runs	The emitted JavaScript	JVM runs bytecode

Important

TypeScript types exist only at compile time. They are completely erased from the JavaScript output. This means types have zero runtime cost — they help you during development and disappear in production. We explore what compiled output looks like — and why Microsoft calls it "compiling" when it's really transpiling — in Building & Running TypeScript.

> **Gen AI & Learning: Why Types Matter for AI Tools**
>
> When you use a coding agent (Copilot, Cursor, Gemini), TypeScript types are one of its most important sources of context. If your function signature says `(user: User) => Response`, the agent knows the shape of the input and the expected output. It can generate correct code, suggest valid property names, and catch mistakes — all because of the type information. Untyped JavaScript forces the agent to guess. Well-typed TypeScript lets it reason precisely. The better your types, the better your AI tools perform.

---

2 Type Annotations

A type annotation tells TypeScript what kind of value a variable, parameter, or return value should hold. The syntax is a colon followed by the type name, placed after the variable name.

2.1 Variable Annotations

The three most common primitive types map directly to what you learned in the JavaScript guide:

```
let name: string = "Alice";
let age: number = 30;
let active: boolean = true;
```

In Java, these would be:

```
String name = "Alice";
int age = 30;
boolean active = true;
```

The key differences:

- The type comes **after** the variable name in TypeScript (`: string`), not before it like Java (`String name`)
- TypeScript's `number` covers both `int` and `double` — there is no integer/floating-point distinction
- TypeScript uses lowercase type names (`string`, `number`, `boolean`), not capitalized class names

⚠ Lowercase Types, Not Wrapper Classes

TypeScript has both `string` and `String`, but they mean different things. Always use the lowercase versions:

```
let name: string = "Alice";  // Correct — primitive type
let name: String = "Alice";  // Wrong — wrapper object (almost
never what you want)
```

This is similar to Java's `int` vs `Integer` distinction, but in TypeScript you should virtually always use the lowercase form.

2.2 Function Parameter and Return Types

Functions are where type annotations provide the most value. Annotate each parameter and the return type:

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

The return type annotation (`: number` after the parameter list) tells both the compiler and other developers what this function produces. Compare with Java:

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

The information is the same — just arranged differently. Java puts the return type before the function name; TypeScript puts it after the parameter list.

Here is a more realistic example — a function you might write in an Express route handler:

```
function formatUserGreeting(name: string, visitCount: number): string {
    if (visitCount === 1) {
        return `Welcome, ${name}! This is your first visit.`;
    }
    return `Welcome back, ${name}! You have visited ${visitCount} times.`;
}
```

If you accidentally return a number or forget the return statement on one code path, TypeScript flags the error immediately.

Try It Yourself

1. Create a file called `types-practice.ts`
2. Write a function `multiply(a: number, b: number): number` that returns the product
3. Try calling it with a string argument: `multiply("five", 3)`
4. Run `npx tsc types-practice.ts` and observe the error message
5. Fix the call and confirm it compiles cleanly

---

3 Type Inference

TypeScript does not always require explicit annotations. When you assign a value at declaration, TypeScript can figure out the type on its own:

```
let count = 5;          // TypeScript infers: number
let greeting = "Hello"; // TypeScript infers: string
let isReady = true;     // TypeScript infers: boolean
```

This is called **type inference**. TypeScript examines the right-hand side of the assignment and locks in the type. After this point, `count` is a `number` — you cannot assign a string to it:

```
let count = 5;
count = "ten"; // Error: Type 'string' is not assignable to type 'number'
```

3.1 When to Annotate vs. When to Let Inference Work

A common question: if TypeScript can figure out types on its own, why annotate at all? The rule of thumb for this course:

Annotate function parameters and return types. Let TypeScript infer local variables.

```
// Good: annotated parameters and return type
function getFullName(first: string, last: string): string {
    const fullName = first + " " + last; // inferred as string — no annotation
needed
    return fullName;
}

// Unnecessary: redundant annotation on a simple assignment
const age: number = 30; // TypeScript already knows this is a number
const age = 30;         // Cleaner — same type safety
```

Why annotate function signatures? Because they are the **boundaries** of your code — the contract between caller and implementation. When someone reads your function signature, they should know exactly what it expects and what it returns without reading the body.

Local variables inside a function are implementation details. Inference keeps them concise without sacrificing safety.

> **Tip**
>
> In Java, you always declare types everywhere because the language requires it. TypeScript gives you the choice. Use annotations at boundaries (function signatures, interface definitions) and rely on inference for the rest. This gives you Java-level safety with less visual noise.

3.2 When Inference Cannot Help

There are situations where TypeScript cannot infer a useful type and you need to annotate explicitly:

```
// Declared without assignment — TypeScript doesn't know the type
let username: string;
username = getUsernameFromRequest(); // assigned later

// Empty array — TypeScript infers 'never[]' without a type annotation
const items: string[] = [];
items.push("first item");
```

If you declare a variable without assigning a value, TypeScript types it as `any` by default (which disables type checking). Adding an explicit annotation prevents this. We cover `any` in detail in Section 9 — The `any` Escape Hatch — for now, just know that `any` turns off type safety and you should avoid it.

---

4 Arrays and Tuples

4.1 Typed Arrays

In Java, you use generics to create typed collections: `ArrayList<String>`. In TypeScript, there are two equivalent syntaxes for typed arrays:

```
const names: string[] = ["Alice", "Bob", "Charlie"];
```

```
const scores: Array<number> = [95, 87, 92];
```

Both mean the same thing. The `string[]` syntax is more common in TypeScript codebases and is what you will see in this course. The `Array<number>` syntax uses generics — familiar from Java — and is useful in more complex type expressions.

Unlike JavaScript, TypeScript prevents mixed-type arrays by default:

```
const names: string[] = ["Alice", 42]; // Error: Type 'number' is not
                                       // assignable to type 'string'
```

In Java, this safety came from generics:

```
ArrayList<String> names = new ArrayList<>();
names.add("Alice");
names.add(42); // Compiler error: incompatible types
```

Same concept, different syntax.

4.2 Tuples

TypeScript has a type that Java does not: **tuples**. A tuple is a fixed-length array where each position has a specific type:

```
const user: [string, number] = ["Alice", 30];
// user[0] is a string (the name)
// user[1] is a number (the age)
```

Tuples are useful when a function needs to return multiple values:

```
function parseIdAndName(input: string): [number, string] {
    const parts = input.split(":");
    return [parseInt(parts[0]), parts[1]];
}

const [id, name] = parseIdAndName("42:Alice");
// id is number, name is string — TypeScript knows the types
```

In Java, you would need to create a class or use `Map.Entry` or a similar wrapper to return two values from a method. TypeScript tuples solve this more concisely. For destructuring tuples and arrays into individual variables, see the Objects, Arrays & Destructuring guide.

⚠ Tuples Look Like Arrays

A tuple `[string, number]` looks like a regular array, but TypeScript enforces both the length and the type at each position. Accessing an index beyond the tuple's length is an error:

```
const pair: [string, number] = ["Alice", 30];
pair[2]; // Error: Tuple type '[string, number]' has no element
at index '2'
```

---

5 Interfaces

Interfaces are one of the most important features in TypeScript — and the place where TypeScript diverges most from Java's approach to types.

5.1 Defining an Interface

An interface defines the shape of an object — what properties it must have and what types those properties must be:

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

Once defined, you can use the interface as a type annotation:

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

If you forget a property or use the wrong type, TypeScript catches it:

```
const bob: User = {
    name: "Bob",
    age: "thirty" // Error: Type 'string' is not assignable to type 'number'
    // Also error: Property 'email' is missing
};
```

In Java, the equivalent requires a class with fields, a constructor, and possibly getters:

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

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

    // getters...
}

User alice = new User("Alice", 30, "alice@example.com");
```

TypeScript's interface plus an object literal achieves the same type safety in far fewer lines.

5.2 Structural Typing — The Biggest Mindset Shift

This is the single most important concept in this guide. Read this carefully.

Java uses **nominal typing**: two objects are compatible only if they share the same class name in their hierarchy. If a method expects a `User`, you must pass an instance of `User` (or a subclass of `User`).

TypeScript uses **structural typing** (also called **duck typing**): two objects are compatible if they have the same shape — the same properties with the same types. The name of the type does not matter. The term "duck typing" comes from the saying: *"If it walks like a duck and quacks like a duck, it's a duck."* TypeScript does not care what an object is called — it cares what properties and methods it has.

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

interface Employee {
    name: string;
    age: number;
    department: string;
}

function greet(user: User): string {
    return `Hello, ${user.name}!`;
}

const employee: Employee = {
    name: "Alice",
    age: 30,
    department: "Engineering"
```

```
};

greet(employee); // Works! Employee has name and age, which is all User
requires
```

The `greet` function asks for a `User` — something with `name: string` and `age: number`. The `Employee` object has both of those properties (plus `department`). TypeScript says: "It has the right shape. It fits."

In Java, this would fail unless `Employee extends User` or both implement a shared interface. The class hierarchy must be declared explicitly. In TypeScript, the relationship is inferred from the structure.

Important

Structural typing is the biggest mindset shift from Java to TypeScript. In Java, you think "what class is this?" In TypeScript, you think "what shape does this have?" This affects how you design interfaces, organize code, and reason about compatibility.

Extra Properties on Object Literals

TypeScript has one exception to its structural flexibility. When you assign an object literal directly to a typed variable, TypeScript performs **excess property checking**:

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

// Direct object literal — excess property check applies
const user: User = {
    name: "Alice",
    age: 30,
    email: "alice@example.com" // Error: 'email' does not exist
in type 'User'
};

// Through a variable — no excess property check
const data = { name: "Alice", age: 30, email:
"alice@example.com" };
const user: User = data; // Works — structural typing allows
extra properties
```

This catches a common class of bugs (typos in property names) while preserving structural typing for variables and function arguments.

5.3 Optional Properties

Not every property is always present. Use `?` to mark a property as optional:

```
interface User {
    name: string;
    age: number;
    email?: string; // Optional — may or may not be present
}

const alice: User = { name: "Alice", age: 30, email: "alice@example.com" }; //
Valid
const bob: User = { name: "Bob", age: 25 };                                  //
Also valid
```

When you access an optional property, TypeScript knows it might be `undefined`:

```
function getEmail(user: User): string {
    return user.email; // Error: Type 'string | undefined' is not
                       // assignable to type 'string'
}

// You must handle the undefined case
function getEmail(user: User): string {
    return user.email ?? "no email provided";
}
```

In Java, any reference can be `null` — but the compiler does not track this for you (unless you use `@Nullable` annotations and a tool like NullAway). TypeScript's optional properties make nullability an explicit, compiler-enforced part of the type.

Try It Yourself

1. Define an interface `Product` with properties `name: string`, `price: number`, and an optional `description?: string`
2. Create two `Product` objects — one with a description, one without
3. Write a function `formatProduct(product: Product): string` that includes the description only if it exists
4. Run `npx tsc` and verify no errors

---

6 Type Aliases

A **type alias** creates a new name for a type. The syntax uses the `type` keyword:

```
type ID = string;
type Coordinate = [number, number];
type StatusCode = 200 | 201 | 400 | 404 | 500;
```

Type aliases are especially useful for union types and complex type expressions (covered in the next section). They give a meaningful name to a type that would otherwise be written inline:

```
// Without a type alias — hard to read
function processResponse(status: 200 | 201 | 400 | 404 | 500): void { ... }

// With a type alias — clear intent
type StatusCode = 200 | 201 | 400 | 404 | 500;
function processResponse(status: StatusCode): void { ... }
```

6.1 `type` vs `interface`

Both `type` and `interface` can describe object shapes, which leads to a common question: when do you use which?

```
// These are functionally equivalent for object shapes
interface UserA {
    name: string;
    age: number;
}

type UserB = {
    name: string;
    age: number;
};
```

The convention in this course (and in most TypeScript codebases):

Use	When
`interface`	Defining the shape of objects — your default choice
`type`	Defining unions, tuples, or giving a name to a non-object type

```
// Use interface for objects
interface User {
    name: string;
    age: number;
}
```

```
// Use type for unions and non-object types
type ID = string | number;
type Pair = [string, number];
type StatusCode = 200 | 201 | 400 | 404 | 500;
```

> **Tip**
>
> If you are defining something that looks like a Java class (a data structure with named properties), use `interface`. If you are defining something that does not exist in Java (a union of possible types, a tuple), use `type`.

---

7 Union Types

A union type allows a variable to hold one of several types. This is a concept with **no direct Java equivalent** — and one of TypeScript's most powerful features.

```
type ID = string | number;

let userId: ID = "abc-123"; // Valid
userId = 42;                // Also valid
userId = true;              // Error: Type 'boolean' is not assignable to type
'ID'
```

The `|` reads as "or" — an `ID` is a `string` *or* a `number`. This is useful in web APIs where an identifier might come from different sources:

```
// Some APIs use numeric IDs, others use string UUIDs
function findUser(id: string | number): User | undefined {
    // Implementation handles both cases
}

findUser(42);           // Valid
findUser("abc-123");    // Valid
findUser(true);         // Error
```

7.1 Type Narrowing

When a variable has a union type, you cannot use type-specific operations on it directly — TypeScript does not know which type it currently holds:

```
function formatId(id: string | number): string {
    return id.toUpperCase(); // Error: Property 'toUpperCase' does not
```

```
                                // exist on type 'number'
}
```

You must **narrow** the type first using a type check:

```
function formatId(id: string | number): string {
    if (typeof id === "string") {
        // Inside this block, TypeScript knows id is a string
        return id.toUpperCase();
    }
    // Here, TypeScript knows id is a number (only option left)
    return id.toString();
}
```

TypeScript tracks the `typeof` check and narrows the type within each branch. This is called **control flow analysis** — the compiler is smart enough to follow your `if` statements and know which type is possible at each point.

Other narrowing techniques you will encounter:

```
// Narrowing with truthiness
function greet(name: string | undefined): string {
    if (name) {
        return `Hello, ${name}!`; // name is string here
    }
    return "Hello, stranger!";
}

// Narrowing with 'in' operator (for objects)
interface Dog { bark(): void; }
interface Cat { meow(): void; }

function speak(animal: Dog | Cat): void {
    if ("bark" in animal) {
        animal.bark(); // TypeScript knows it's a Dog
    } else {
        animal.meow(); // TypeScript knows it's a Cat
    }
}
```

7.2 Why Java Does Not Have This

In Java, a variable has exactly one type (or a type in its class hierarchy). If a method could accept either a `String` or an `Integer`, you would use method overloading:

```
public String formatId(String id) {
    return id.toUpperCase();
}

public String formatId(int id) {
```

```
    return String.valueOf(id);
}
```

Or you would use a common supertype like `Object` and cast:

```
public String formatId(Object id) {
    if (id instanceof String) {
        return ((String) id).toUpperCase();
    }
    return id.toString();
}
```

TypeScript's union types are more precise — `string | number` is tighter than `Object` — and the narrowing is automatic (no casting required).

Try It Yourself

1. Define a type alias: `type Result = string | number | boolean`
2. Write a function `describe(value: Result): string` that returns:
   - `"text: <value>"` if it is a string
   - `"number: <value>"` if it is a number
   - `"flag: <value>"` if it is a boolean
3. Use `typeof` to narrow the type in each branch
4. Call the function with all three types and log the results

---

8 Functions in TypeScript

You have already seen basic function type annotations. This section covers the patterns you will use most often in Express route handlers and utility functions.

8.1 Typed Parameters and Return Values

A fully typed function specifies the type of every parameter and the return type:

```
function createGreeting(name: string, age: number): string {
    return `Hello, ${name}! You are ${age} years old.`;
}
```

Arrow functions follow the same pattern:

```
const createGreeting = (name: string, age: number): string => {
    return `Hello, ${name}! You are ${age} years old.`;
};
```

8.2 Optional Parameters

In Java, if you want a method to accept different numbers of arguments, you overload it:

```
public String greet(String name) {
    return greet(name, "Hello");
}

public String greet(String name, String greeting) {
    return greeting + ", " + name + "!";
}
```

TypeScript uses optional parameters instead — a single function with `?` on the parameters that may be omitted:

```
function greet(name: string, greeting?: string): string {
    const g = greeting ?? "Hello"; // Use "Hello" if greeting is undefined
    return `${g}, ${name}!`;
}

greet("Alice");          // "Hello, Alice!"
greet("Alice", "Hey");   // "Hey, Alice!"
```

Optional parameters must come after required parameters — you cannot put `greeting?` before `name`.

8.3 Default Values

Default values take optional parameters one step further. Instead of checking for `undefined` yourself, you provide a fallback in the parameter list:

```
function greet(name: string, greeting: string = "Hello"): string {
    return `${greeting}, ${name}!`;
}

greet("Alice");          // "Hello, Alice!"
greet("Alice", "Hey");   // "Hey, Alice!"
```

This is cleaner than the optional parameter version because you do not need the `??` fallback logic. The function body can use `greeting` directly — it will always be a `string`.

8.4 Overloading vs. Optional/Default Params

In Java, you write multiple method signatures — the compiler picks the right one. In TypeScript, you write one function with optional or default parameters. This is a deliberate design difference.

Java Approach	TypeScript Approach
Method overloading — multiple signatures	One function with optional/default params
Compiler dispatches to correct overload	Runtime checks or defaults handle variants
Can have completely different parameter types	Optional params must share a single signature

TypeScript does support function overloading for advanced cases, but it is rarely needed. Optional and default parameters handle the vast majority of situations you will encounter.

8.5 Function Types as Parameters

In Java, passing behavior to a method requires an interface and an anonymous class (or a lambda in Java 8+). In TypeScript, you can specify a function type directly in the parameter list:

```
function processItems(items: string[], callback: (item: string) => string):
string[] {
    return items.map(callback);
}

const uppercased = processItems(["hello", "world"], (item) =>
item.toUpperCase());
// ["HELLO", "WORLD"]
```

The type `(item: string) => string` means "a function that takes a `string` and returns a `string`." TypeScript enforces this — if you pass a function with the wrong signature, the compiler catches it.

This pattern appears everywhere:

- **Array methods:** `filter`, `map`, `find` all accept function parameters
- **Express middleware:** `(request: Request, response: Response, next: NextFunction) => void`
- **Event handlers:** `(event: Event) => void`

For complex function types, you can use a **type alias** to keep signatures readable:

```
type Validator = (value: string) => boolean;

function validateAll(values: string[], validator: Validator): boolean {
    return values.every(validator);
}

const isNonEmpty: Validator = (value) => value.length > 0;
validateAll(["hello", "world"], isNonEmpty); // true
```

Try It Yourself

1. Write a function `buildUrl(base: string, path: string, port: number = 3000): string`
2. It should return strings like `"http://localhost:3000/users"`
3. Call it with and without the `port` argument
4. Try passing a string for `port` and see what TypeScript says

---

9 The `any` Escape Hatch

TypeScript has a special type called `any` that disables all type checking for a value:

```
let data: any = "hello";
data = 42;              // No error
data = true;            // No error
data.nonexistent.method(); // No error at compile time — crashes at runtime
```

With `any`, you are back to JavaScript — no type safety, no autocomplete, no error checking. TypeScript trusts you completely and checks nothing.

9.1 Why `any` Exists

You will encounter `any` in a few legitimate situations:

- **Untyped third-party libraries.** Some npm packages do not include TypeScript type definitions. Their APIs return `any` because TypeScript does not know what shape the data has.
- **Gradual migration.** When converting a JavaScript codebase to TypeScript, developers use `any` as a temporary placeholder while adding types incrementally.

- **Quick prototyping.** During early experimentation, you might use `any` to avoid fighting the type system before you know what shape your data will take.

9.2 Why You Should Avoid `any`

In this course, using `any` is almost always a mistake. It means you have given up on type safety for that value — and bugs will hide there.

```
// Bad: any spreads through your code
function processData(data: any) {
    const result = data.items.map((item: any) => item.name);
    // No type checking anywhere in this function
    // Typos, missing properties, wrong types — all invisible
    return result;
}

// Good: define the shape
interface DataResponse {
    items: Array<{ name: string }>;
}

function processData(data: DataResponse) {
    const result = data.items.map(item => item.name);
    // TypeScript verifies everything — items exists, name exists, types match
    return result;
}
```

> ⚠ **The `any` Virus**
>
> `any` is contagious. If a function accepts `any`, its return value is also `any`. Any variable that touches `any` becomes `any`. A single `any` in a chain of function calls can disable type checking across your entire data flow. Treat `any` as a code smell — when you see it, ask whether you can replace it with a real type.

9.3 `unknown` — The Safe Alternative

TypeScript provides `unknown` as a safe alternative to `any`. Like `any`, a variable typed `unknown` can hold any value. Unlike `any`, you **cannot use it** without first narrowing the type:

```
let data: unknown = fetchExternalData();

// With any — TypeScript trusts you blindly
let dataAny: any = fetchExternalData();
dataAny.name.toUpperCase(); // No error — crashes at runtime if wrong

// With unknown — TypeScript forces you to check first
data.name; // Error: 'data' is of type 'unknown'
```

```
if (typeof data === "object" && data !== null && "name" in data) {
    console.log(data.name); // Now TypeScript knows data has a 'name' property
}
```

The rule: use `unknown` when you genuinely do not know the type (e.g., parsing JSON from an external API). Then narrow it before using it. This gives you safety without lying to the compiler.

Type	Assign any value?	Use without checking?	Safety
`any`	Yes	Yes	None — type checking disabled
`unknown`	Yes	No — must narrow first	Full — forces validation

Tip

When you are tempted to type something as `any`, try `unknown` instead. If you cannot make the code work with `unknown`, that is a sign you need to define an interface for the data rather than bypassing the type system.

---

10 Summary

Concept	Key Point
TypeScript's purpose	Catches errors at compile time, not runtime — brings Java-like safety to JavaScript
Type annotations	`: type` after variable/parameter names — annotate function signatures, infer locals
Type inference	TypeScript deduces types from assigned values — reduces boilerplate without losing safety
Arrays	`string[]` or `Array<string>` — typed, homogeneous collections

Concept	Key Point
Tuples	`[string, number]` — fixed-length arrays with per-position types
Interfaces	Define object shapes — `interface User { name: string; age: number; }`
Structural typing	Compatibility based on shape, not class name — the biggest shift from Java
Type aliases	`type ID = string | number` — names for unions and complex types
Union types	`string | number` — a value that could be one of several types (no Java equivalent)
Type narrowing	`typeof`, `in`, truthiness checks — proves which type a union currently holds
Optional properties	`email?: string` — explicitly marks what may be undefined
Optional/default params	`greeting?: string` or `greeting = "Hello"` — replaces Java's method overloading
`any`	Disables type checking — avoid it; it spreads and hides bugs
`unknown`	Safe alternative to `any` — must narrow before use

11 References

Official Documentation:

- TypeScript Handbook — Everyday Types — Covers type annotations, interfaces, unions, and type narrowing
- TypeScript Handbook — Narrowing — Deep dive into control flow analysis and type guards
- TypeScript Handbook — Object Types — Interfaces, optional properties, and structural typing

- MDN Web Docs — typeof operator — JavaScript's runtime type checking used for narrowing

Tutorials:

- TypeScript Playground — Browser-based editor for experimenting with TypeScript without any setup

---

12 Further Reading

External Resources

- TypeScript Handbook — Type Inference — How TypeScript deduces types automatically
- TypeScript Handbook — Interfaces vs Types — Official guidance on when to use each
- TypeScript Handbook — Functions — Optional parameters, overloads, and generic functions
- TypeScript Deep Dive — Type Guard — Community guide with additional narrowing patterns

---

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