TYPESCRIPT

PREMCHANDNAIDU.G

Introduction to TypeScript

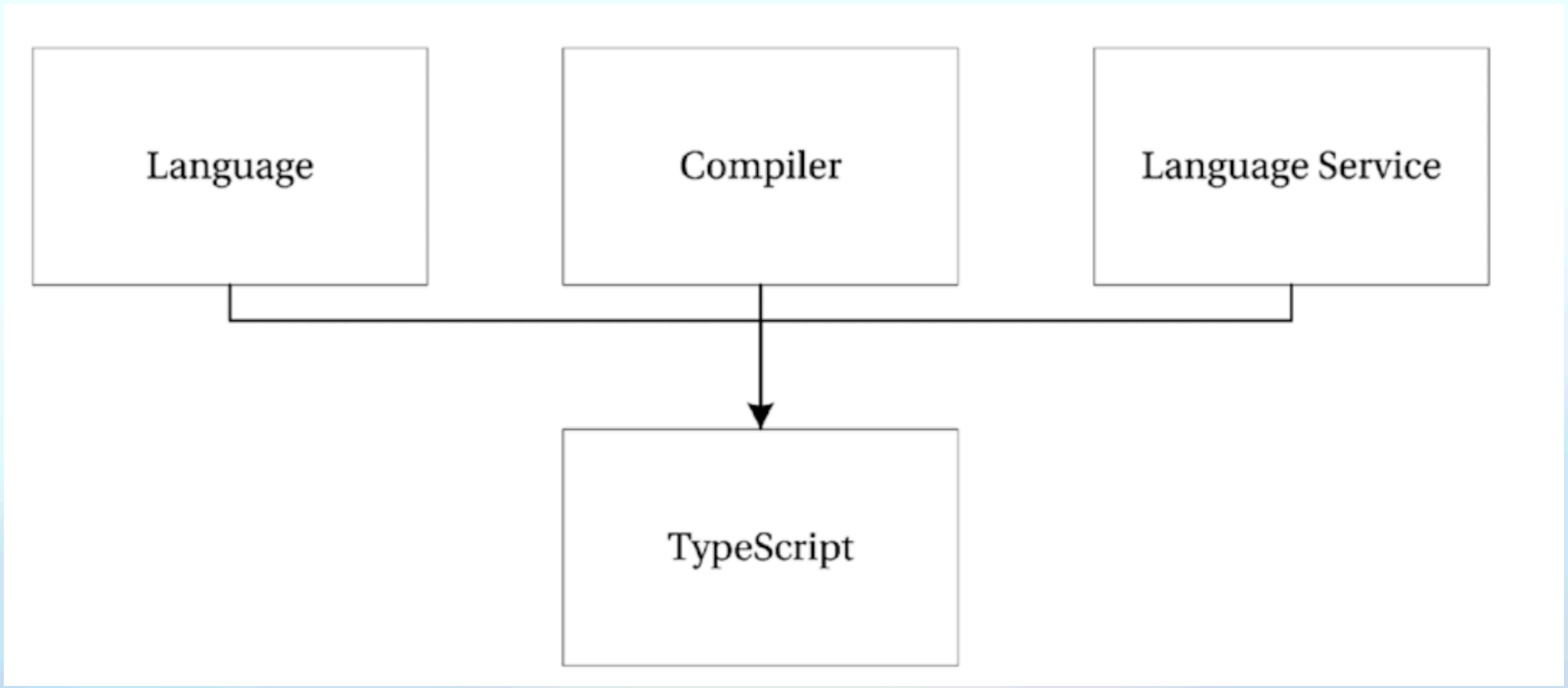
What is type script?

TypeScript is an application-scale programming language that provides early access to proposed new JavaScript features and powerful additional features like static type checking. You can write TypeScript programs to run in web browsers or on servers and you can reuse code between browser and server applications.

TypeScript solves many problems in JavaScript, but it respects the patterns and implementation of the underlying JavaScript language, for example, the ability to have dynamic types.

You can use many integrated development environments with TypeScript, with several providing first-class support including type checking and autocompletion that will improve your productivity and helpeliminate mistakes at design time.

- TypeScript is a language, a compiler, and a language service.
- You can paste existing JavaScript into your TypeScript program.
- Compiling from TypeScript to JavaScript is known specifically as transpiling.
- TypeScript is not the only alternative way of writing JavaScript, but it has gainedincredible traction in its first five years.


Language
Compiler
Language Service
TypeScript

1. Language

The language part of TypeScript includes its **syntax, keywords, and type system**.
It allows developers to define data types, interfaces, and structures in their programs.
By using types, the programmer clearly describes how data should be used, which helps avoid errors and makes the code easier to understand and maintain.

2. Compiler

The TypeScript compiler is responsible for **converting TypeScript code into JavaScript**.
During this process, it checks the code for type-related errors and gives warnings if it finds any issues.
The compiler removes all type information (type erasure) and can also perform additional tasks such as combining files, generating source maps, and transforming modern syntax into compatible JavaScript.

3. Language Service

The language service provides **intelligent support for developers while writing code**.
It uses the type information from the program to offer features like auto-completion, type hints, error highlighting, and refactoring tools.
This service improves productivity and helps developers write correct and clean code faster.

Why typescript is needed?

More Secure
TypeScript adds type safety to JavaScript, which helps prevent common programming mistakes. By catching errors early, it reduces the chances of unexpected behavior in applications, making the code more reliable and secure.

Readable and Maintainable Code
With clear type definitions and structured code, TypeScript makes programs easier to read and understand. This improves maintainability, especially when working in teams or on long-term projects.

Early Error Detection

TypeScript detects errors at the **coding stage** by providing warnings, suggestions, and compile-time errors. This helps developers fix problems before the application runs, saving time and effort.

Better Tooling Support

TypeScript offers excellent support in modern editors like VS Code, including auto-completion, type hints, refactoring, and intelligent suggestions, which improve developer productivity.

Suitable for Large-Scale Applications

Because of its strong typing, scalability, and maintainability, TypeScript is widely used in **large and secure applications** such as enterprise systems, financial platforms, and complex web applications.

Ts vs js

Feature	**JavaScript (JS)**	**TypeScript (TS)**
Type System	Dynamic typing	Static typing
Error Detection	Runtime errors	Compile-time errors
Syntax	Simple, flexible	Slightly stricter
Code Safety	Less safe	More safe due to types
Readability	Harder in large apps	More readable & structured
Maintainability	Difficult for big projects	Easy to maintain
Tooling Support	Basic	Excellent (auto-complete, hints)
Scalability	Not ideal for large apps	Best for large-scale apps
Compilation	Runs directly	Compiled to JavaScript
Learning Curve	Easy to start	Slightly more learning

Js problems->typescript solutions

Each and every value in JavaScript has a set of behaviors you can observe from running different operations. That sounds abstract, but as a quick example, consider some operations we might run on a variable named message.

```
// Accessing the property 'toLowerCase'
// on 'message' and then calling it
message.toLowerCase();
// Calling 'message'
message();
```

If we break this down, the first runnable line of code accesses a property called toLowerCase and then calls it. The second one tries to call message directly.

But assuming we don't know the value of message - and that's pretty common - we can't reliably say what results we'll get from trying to run any of this code.

The behavior of each operation depends entirely on what value we had in the first place. Is message callable?

Does it have a property called toLowerCase on it?

If it does, is toLowerCase even callable?

If both of these values are callable, what do they return?

Let's say message was defined in the following way.

```
const message = "Hello World!”;
```

As you can probably guess, if we try to run message.toLowerCase(), we'll get the same string only in lower-case.

What about that second line of code? If you're familiar with JavaScript, you'll know this fails with an
exception:

```
TypeError: message is not a function
```

It'd be great if we could avoid mistakes like this.
When we run our code, the way that our JavaScript runtime chooses what to do is by figuring out the type of the value - what sorts of behaviors and capabilities it has. That's part of what that

TypeError is alluding to - it's saying that the string "Hello World!" cannot be called as a function.

Real world use cases of Type Script

- 1.
- 2.
- 3.
- 4.
- 5.

Steps to install Node js and TypeScript

- Step-1: install node.js from browser
- Step-2: During installation create a folder for installing and writing a code
- Step-3: After installing check these commands
    - Node -v
    - Npm -v
    - Npx -v

- Step-4: Installing type script globall
  - Npm install -g typescript
- Step-5: after installing check installation done or not ny using
- Tsc -v
- Step-6: open folder in vs code
- Step-7: create a new .ts file

What is tsc?

`tsc` stands for **TypeScript Compiler**.

It is a **command-line tool** used to **convert TypeScript code into JavaScript**.

What does `tsc` do?

- Compiles `.ts` files into `.js` files
- Checks for **type errors** in the code
- Removes type information (type erasure)
- Shows compile-time **errors and warnings**
- Uses settings from `tsconfig.json`

Why `tsc` is important

Without `tsc`, browsers cannot understand TypeScript.
`tsc` ensures your TypeScript code is **safe, clean, and ready to run as JavaScript**.

Lab-1

Task 1: Identify JavaScript Problems

1. Write a small JavaScript example that:
   - Assigns a number to a variable
   - Later assigns a string to the same variable
2. Observe that JavaScript allows this without errors.
3. Note down:
   - What problem this can cause in large application

Lab-2

Task 2: Solve the Same Problem Using TypeScript

1. Create a file named:
   `app.ts`
2. Declare variables using **TypeScript types** (`number`, `string`).
3. Try assigning a wrong type and observe the compiler error.

Lab-3

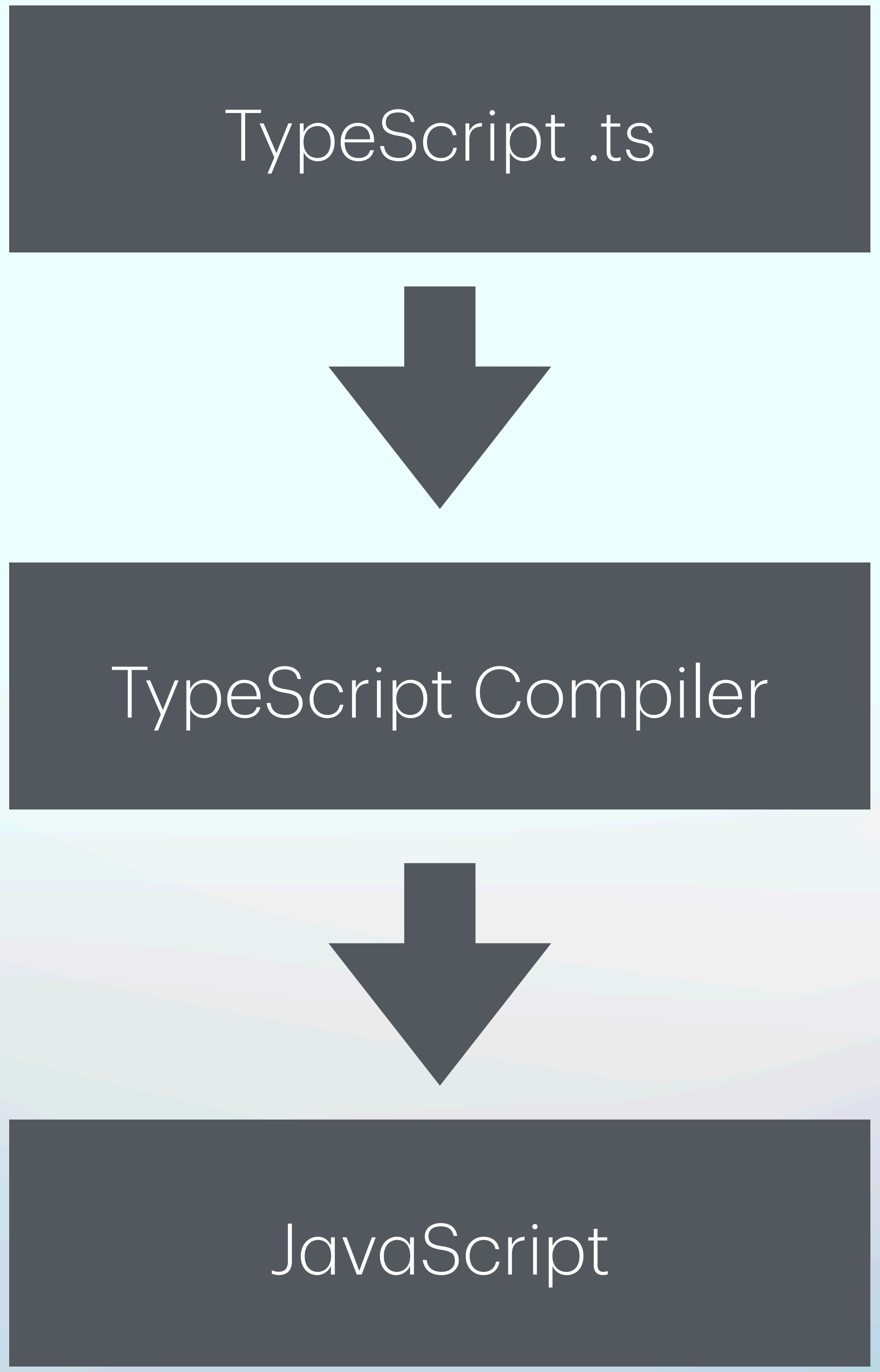
Task 3: TypeScript vs JavaScript Comparison

Create a small table or note showing:

- How JavaScript handles types
- How TypeScript handles types
- Which one gives errors earlier and why

How TypeScript works (TS → JS)

TypeScript does not run directly in the browser.
First, TypeScript code is **compiled into JavaScript** using the TypeScript compiler.



During compilation:

- Types are checked
- Errors are reported
- JavaScript output is generated

Steps to install vite with ts

- Step-1: npx create-vite@latest
- Step-2: project name
- Step-3: package name
- Step-4: select vanilla
- Step-5: select ts
- Step-6: yes

Executing ts file

- Step-1: open integrated terminal
- Step-2: use command tsc filename.ts
- Step-3:After compilation, a javascript file will be generated
    - Filename.ts-> filename.js
- Step-4: Run the generated javascript file by using node.js
    - Node filename.js

Type inference

- What is type inference?
- How to implement type inference?
- Where type interface is used?

`tsconfig.json`

`tsconfig.json` is a **configuration file** used by the TypeScript compiler (`tsc`).
It tells TypeScript **how to compile your project**, which files to include, and which rules to follow.

When `tsc` finds a `tsconfig.json`, it treats that folder as a **TypeScript project**

Why `tsconfig.json` is important

- Keeps **consistent compiler settings** across the project
- Enables **strict type checking**
- Controls **output location** and JavaScript version
- Helps in **large and team-based projects**

Common `compilerOptions`

Target

The `target` option specifies the **JavaScript version** that the TypeScript compiler should generate.
It ensures that the output JavaScript is compatible with the environment where the code will run, such as older browsers or modern platforms.

Module

The `module` option defines the **module system** used in the generated JavaScript.
It controls how files import and export code, which is important for bundlers and runtime environments like Node.js or browsers.

Strict

The `strict` option enables **strong type-checking rules** across the project.
When enabled, TypeScript performs deeper checks to identify potential errors early, improving code reliability and quality.
It is highly recommended for professional and large-scale applications.

outDir

The `outDir` option specifies the **output folder** where all compiled JavaScript files are placed.
This helps keep the source TypeScript files separate from the generated JavaScript, maintaining a clean project structure.

rootDir

The `rootDir` option defines the **main source folder** for TypeScript files.
It tells the compiler where the original code is located so that the folder structure is preserved in the output directory.

Strict mode

- 1.What is Strict mode?
- 2.Why it is important?

Lab-4

- Create a react application using type script
- Create a new tsx component
- Connect to app.js
- Execute conditional rendering example

Basic Data Types

- 1.Primitive data types
- 2.Utility data types
- 3.Collection Data types
- 4.Object based data types

Group	Purpose	Types
Primitive	Single values	number, string, boolean, null, undefined
Special / Utility	Control & safety	any, unknown, void, never
Collection	Multiple values	array, tuple
Structured	Complex data	object, enum

1.primitive data types

Used to store single simple values

- **number** – used to store numeric values
- **string** – used to store text / character values
- **boolean** – used to store `true` or `false` values
- **null** – used to represent an intentional empty value
- **undefined** – used when a variable is declared but not assigned any value

2.utility data types

Used for flexibility, safety, or special behavior

- **any**
  → Used when the data type is **not known in advance**, such as **API responses**, **third-party libraries**, or **dynamic user input**
- **unknown**
  → Used when the data type is **unknown but must be checked** before using it (safer than `any`), commonly for **user input** or **external data**
- **void**
  → Used for **functions that perform an action but do not return any value**, such as logging, saving data, or showing alerts
- **never**
  → Used for **functions that never complete normally**, such as **error handling**, **infinite loops**, or functions that always throw an error

Explicit vs implicit typing

- Give minimum of 4 differences
- 1.
- 2.
- 3.
- 4.

Arrays, Tuples & Objects

- 1.Arrays:

→ Used to store **multiple values of the same type**
→ Commonly used for **lists** (users, products, marks)
let Students_marks:number =[1,2,3,4,5]

- 2. Tuples

→ Used to store a **fixed number of values with fixed types and order**
→ Commonly used when **structure is known** (id + name, coordinates)
let Students_marks:{number, string, number} =[1,"Ravi",34]

Object

→ Used to store **key–value pairs**
→ Used to represent **real-world entities** (user, product, employee)

```
Let Student:{Id:number,name:String,Age:number}={
Id:1,
Name:"raju",
Age:34
}
```

Lab-5

Type Aliases & Union Types

Type aliases

Union types (|)

Literal types

Type narrowing

Practical examples

Functions in TypeScript

Function return types

Optional & default parameters

Arrow functions

Function type annotations

Real-world function example