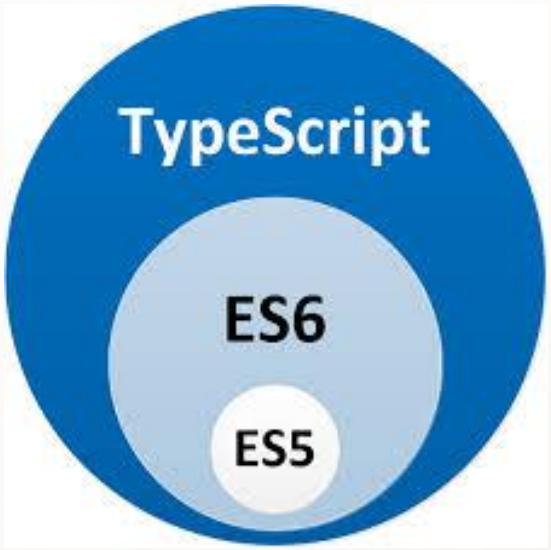
Application Delivery Fundamentals



Parameswari Ettiappan

Typescript

Goals

- *Introduction to TypeScript*
- *Using Visual Studio Code for TypeScript Development*
- *TypeScript Language Basics*
- *TypeScript Type System*
- *Functional Programming*
- *Object-Oriented Programming*
- *Asynchronous Programming*
- *Generics*
- *Namespaces and Modules*
- *Unit Testing*

History of JavaScript

- **1. Birth of JavaScript (1995)**
  - **Inventor**: Brendan Eich at **Netscape Communications**.
  - **Original name**: Mocha → later renamed to **LiveScript**, and finally **JavaScript** (for marketing reasons, to ride on Java's popularity).
  - **Purpose**: Add interactivity to static web pages (e.g., form validation, dynamic content).

History of JavaScript

- **2. Standardization (1997)**
  - **ECMAScript**: JavaScript was standardized by **ECMA International** under **ECMAScript (ES)**.
  - **First version**: ECMAScript 1 (1997).
  - Important for creating consistent JavaScript behavior across browsers.

History of JavaScript

- **3. Browser Wars (Late 1990s–2000s)**
- JavaScript implementations varied widely (e.g., Netscape's SpiderMonkey vs Internet Explorer's JScript).
- This led to inconsistencies in how JavaScript behaved, sparking the need for standard compliance.

History of JavaScript

- **4. AJAX and Web 2.0 (2004–2010)**
- AJAX (Asynchronous JavaScript and XML) enabled dynamic, single-page applications (e.g., Gmail, Google Maps).
- Shifted JavaScript from simple scripting to full-blown **application logic**.

History of JavaScript

- **5. Rise of Frameworks (2010s)**
- Libraries like **jQuery**, **AngularJS**, **React**, **Vue.js** revolutionized frontend development.
- Developers could build **complex, maintainable, and modular** applications.

History of JavaScript

- **6. Node.js (2009)**
- JavaScript on the **server-side** via the V8 engine (by Google).
- Allowed full-stack development with one language — **JavaScript everywhere**.

History of JavaScript

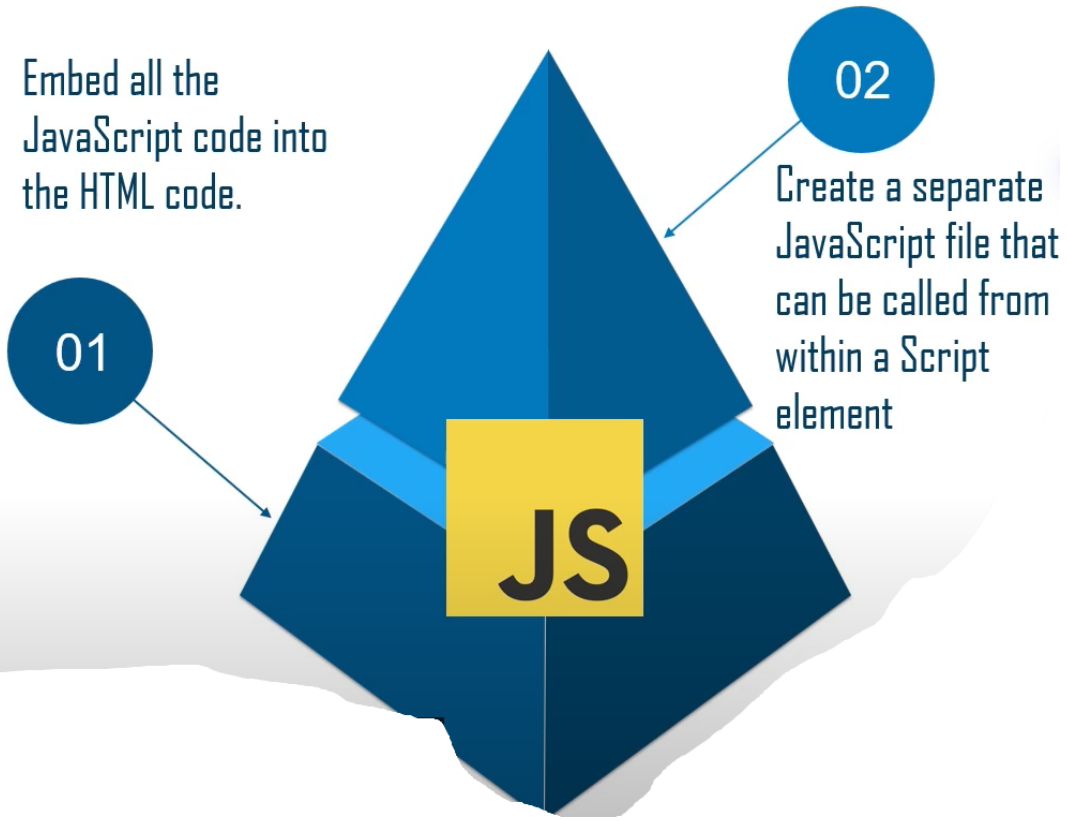
- **7. Modern JavaScript (2015 onward)**
  - **ECMAScript 6 (ES6/ES2015): Introduced let, const, arrow functions, classes, promises, modules, etc.**
  - **Regular updates (ES7, ES8...) enhance performance, syntax, and capabilities.**

JavaScript

JavaScript is a high level, interpreted, programming language used to make web pages more **interactive**. It let's you implement complex and beautiful things/design on **web pages.**

TS

HOW TO USE JAVASCRIPT?



Features of JavaScript

Scripting language & not Java

e!

Object-based

It runs in a browser

Features of JavaScript

- It is a Scripting Language and has nothing to do with Java. Initially, It was named Mocha, then changed to Live Script and finally it was named as JavaScript.

- JavaScript is an object-based programming language that supports polymorphism, encapsulation, and inheritance as well.

- You can run JavaScript not only in the browser but also on the server and any device which has a JavaScript Engine.

What can JavaScript do?

JavaScript is used to create beautiful **web pages** and **applications**. It is mostly used to make your web look alive and adds variety to the page.

It is also used in **smartwatches**. An example of this is the popular smartwatch maker called Pebble that has created a small JavaScript Framework called Pebble.js.

JavaScript is also used to make **Games**. A lot of developers are building small-scale games and apps using JavaScript.

Most popular websites like Google, Facebook, Netflix, Amazon, etc make use of JavaScript to build their **websites**.

Importance of JavaScript Today

- **1. Ubiquity**
- **Runs in all modern browsers** by default.
- Powers **nearly every interactive feature** on the web.
- **2. Full-Stack Capabilities**
- With tools like **Node.js**, JavaScript is used for:
  - Frontend (React, Angular, Vue)
  - Backend (Express.js, NestJS)
  - Databases (e.g., MongoDB drivers)
  - APIs and microservices

Importance of JavaScript Today

- **3. Community and Ecosystem**
- Largest number of packages (via **npm**).
- Massive open-source community and support.
- **4. Versatility**
- Web apps, mobile apps (React Native), desktop apps (Electron), IoT, AI, and game development.
- **5. Foundation for Modern Web Apps**
- SPAs (Single Page Applications)
- PWA (Progressive Web Apps)
- Real-time features (sockets, streaming)

Importance of JavaScript Today

- **6. Career Relevance**
- In-demand skill in web development.
- Common requirement in full-stack and frontend developer roles.

Summary

Aspect	Description
Invented	1995 by Brendan Eich (Netscape)
Standardized As	ECMAScript (ECMA-262)
Key Milestones	AJAX, Node.js, ES6, Frameworks (React, Angular)
Uses	Frontend, Backend, Mobile, Desktop
Importance	Universal for modern web development

Strength of Java Script

#	Strength	Explanation
1	**Universally Supported**	Runs in all modern web browsers without plugins.
2	**Client-Side Execution**	Reduces server load and provides instant feedback to users.
3	**Versatile (Full-Stack)**	Can be used for frontend (React, Angular) and backend (Node.js).
4	**Rich Ecosystem**	npm has millions of libraries and tools to accelerate development.
5	**Interactive UIs**	Enables dynamic HTML manipulation (DOM), animations, event handling.
6	**Asynchronous Programming**	Promises and async/await simplify working with APIs and real-time apps.
7	**Large Community**	Massive developer base, extensive tutorials, and active open-source support.
8	**Constantly Evolving**	ECMAScript updates add modern features like classes, modules, optional chaining, etc.
9	**Cross-Platform**	JavaScript can build mobile apps (React Native), desktop apps (Electron), and IoT apps.

Weaknesses of JavaScript

#	Weakness	Explanation
1	**Security Risks**	Exposed to cross-site scripting (XSS), code injection, etc. because it runs on the client.
2	**Browser Inconsistencies**	Historically had differences in how browsers interpreted JS, though ES standards have improved this.
3	**Weak Typing**	No strict types by default (until TypeScript); can lead to unpredictable behavior ("5" + 1 = "51").
4	**Performance Limitations**	Slower than compiled languages like C++, Rust, or even Java in CPU-intensive tasks.
5	**Callback Hell (historically)**	Before async/await, managing multiple callbacks led to messy and hard-to-read code.
6	**Can Be Misused Easily**	Simple syntax leads beginners to write poor-quality or insecure code if not careful.
7	**No Multithreading (Traditionally)**	JavaScript runs in a single thread (except Web Workers), limiting concurrency.
8	**Tooling Complexity**	Modern JS development (with Babel, Webpack, TypeScript) has a steep learning curve.

Limitations of JavaScript

#	Limitation	Description
1	**Single-threaded Execution**	JavaScript uses a single-threaded event loop model, which can be a bottleneck for CPU-heavy tasks. Multithreading is limited (Web Workers exist but are not seamless).
2	**No File System Access (in browser)**	JavaScript in the browser cannot directly read/write to the user's file system (for security reasons).
3	**Security Vulnerabilities**	JavaScript is susceptible to XSS (Cross-Site Scripting), CSRF, and other client-side attacks due to its openness and execution in the browser.
4	**Weak Typing**	Dynamically typed: types are not enforced, leading to unpredictable behavior and hard-to-catch bugs ("5" - 2 = 3 but "5" + 2 = "52").
5	**Client Dependency**	If JavaScript is disabled in the user's browser (or blocked by extensions), the application may break or not function.
6	**No Direct Access to OS or Hardware APIs**	JavaScript cannot access OS-level APIs (processes, threads, device drivers) — except in controlled environments like Node.js or Electron.

Limitations of JavaScript

7	**Browser Incompatibility (historically)**	Though mostly resolved today, JavaScript used to behave differently across browsers, requiring polyfills and fallbacks.
8	**Debugging Complexity**	Although modern dev tools are powerful, debugging asynchronous code (promises, callbacks, event handlers) can still be tricky.
9	**Lack of Strong Concurrency Model**	Unlike languages like Go or Rust, JavaScript lacks built-in concurrency primitives like goroutines or threads. Async behavior is cooperative, not parallel.
10	**Heavy Tooling Overhead (Modern JS)**	Building modern JS apps requires configuring transpilers (Babel), bundlers (Webpack/Vite), linters, and compilers (TypeScript), increasing complexity.
11	**Limited Numeric Precision**	Uses IEEE-754 64-bit floating-point format — problematic for very large integers or high-precision calculations (use BigInt carefully).
12	**Global Namespace Pollution (in older code)**	Early JavaScript lacked modules, so scripts could easily overwrite each other's global variables/functions. (ES6 modules now fix this.)

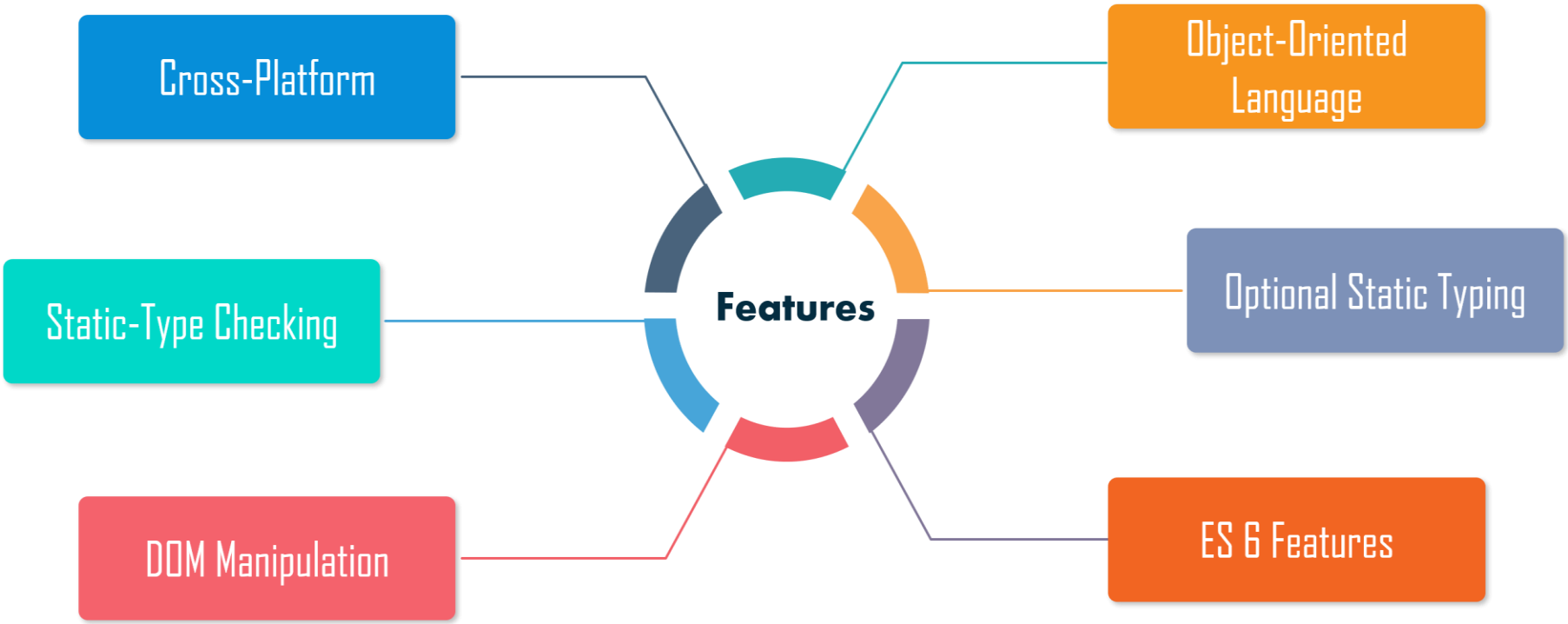
TypeScript

- TypeScript is a typed superset of JavaScript that compiles to plain JavaScript.
- TypeScript is pure object-oriented with classes, interfaces and statically typed programming languages like C# or Java.
- It requires a compiler to compile and generate in JavaScript file.
- Basically, TypeScript is the ES6 version of JavaScript with some additional features.

TypeScript

- A TypeScript code is written in a file with .ts extension and then compiled into JavaScript using the compiler.
- You can write the file in any code editor and the compiler needs to be installed on your platform.
- After the installation, the command tsc <filename>.ts compiles the TypeScript code into a plain JavaScript file.

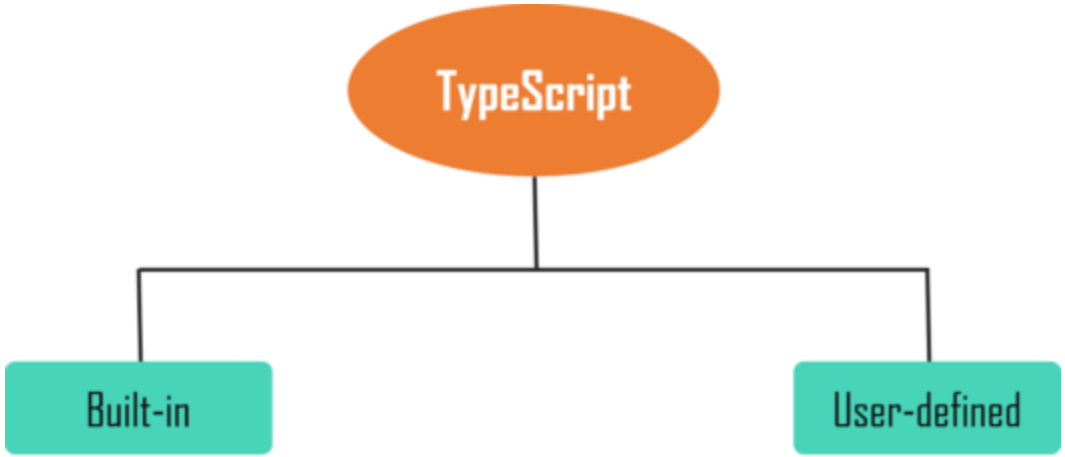
Features of TypeScript



Features of TypeScript

- Cross-Platform: The TypeScript compiler can be installed on any Operating System such as Windows, MacOS, and Linux.
- Object-Oriented Language: TypeScript provides features like Classes, Interfaces, and Modules. Thus, it can write object-oriented code for client-side as well as server-side development.
- Static type-checking: TypeScript uses static typing and helps type checking at compile time. Thus, you can find errors while writing the code without running the script.
- Optional Static Typing: TypeScript also allows optional static typing in case you are using the dynamic typing of JavaScript.
- DOM Manipulation: You can use TypeScript to manipulate the DOM for adding or removing elements.
- ES 6 Features: TypeScript includes most features of planned ECMAScript 2015 (ES 6, 7) such as class, interface, Arrow functions, etc.

Types of TypeScript



It can be classified into two types such as:
•**Built-in**: This includes number, string, boolean, void, null and undefined.
•**User-defined**: It includes Enumerations (enums), classes, interfaces, arrays, and tuple.

Advantages of Typescript

#	Advantage	Explanation
1	**Static Typing**	TypeScript introduces static types (string, number, boolean, etc.), catching errors **at compile time** instead of runtime.
2	**Improved Code Quality & Readability**	Types act as **self-documentation**, making code easier to understand and maintain.
3	**Better IDE Support**	Editors like VS Code offer **autocompletion**, **intellisense**, and **real-time type checking** for TypeScript, improving developer productivity.
4	**Early Bug Detection**	TypeScript's compile-time checks catch common bugs (e.g., null/undefined access, wrong function arguments) before deployment.
5	**Object-Oriented Features**	TypeScript supports classes, interfaces, abstract classes, access modifiers (private, protected), and more — useful for large-scale apps.
6	**Code Scalability**	TypeScript is ideal for large codebases due to modularity, type contracts, and better tooling for refactoring.

Advantages of Typescript

7	**Optional Type Annotations**	You can gradually add types — TypeScript doesn't force strict typing everywhere (any, type inference).
8	**Enhanced Tooling Ecosystem**	Works seamlessly with modern frontend tools (Webpack, Babel, ESLint) and frameworks (Angular, React, Vue).
9	**Support for Latest ECMAScript Features**	TypeScript compiles to plain JavaScript and allows you to use **future JavaScript (ESNext)** features safely.
10	**Improved Collaboration**	Strong types and contracts make it easier for teams to work together with fewer misunderstandings.
11	**Open Source and Backed by Microsoft**	Actively maintained, well-documented, and widely supported in enterprise development.

Typescript Design Goals

#	Design Goal	Explanation
1	**Static Typing for JavaScript**	Provide optional static typing to catch errors **at development time** and improve tooling, without changing the JavaScript runtime.
2	**Preserve JavaScript Compatibility**	TypeScript is a **strict superset of JavaScript**, meaning any valid JavaScript code is valid TypeScript code.
3	**Embrace Existing JavaScript Ecosystem**	TypeScript integrates with **existing JS libraries, tools, and runtimes**, including npm, React, Node.js, etc.
4	**Support for Modern JavaScript Features**	Allow developers to use the **latest ECMAScript features** (e.g., async/await, modules) even before they're fully supported in browsers or Node.js.
5	**Gradual Adoption**	TypeScript enables **incremental migration**: teams can start by renaming .js files to .ts and gradually add types (any is allowed initially).

Typescript Design Goals

6	**Strong Tooling Support**	Enhance developer productivity with **powerful editor features** like autocompletion, inline documentation, and refactoring support.
7	**Scalable for Large Applications**	Designed to support **enterprise-scale** projects through interfaces, namespaces, modules, and clear type contracts.
8	**Erase Types at Compile Time**	TypeScript compiles down to **clean, readable JavaScript**, and all types are **stripped at compile time**, ensuring no runtime overhead.
9	**Maintain Familiar JavaScript Syntax**	Avoid creating an entirely new language — TypeScript extends JS syntax with minimal disruption.
10	**Enable Richer Code Navigation & Refactoring**	Type awareness allows IDEs to provide **smarter code navigation, go-to-definition, and refactorings**, making development more efficient.

How to install TypeScript

- There are two main ways to install TypeScript tools such as:
  - Via npm (Node.js Package Manager) command-line tool
  - npm install -g typescript
  - By installing TypeScript via Visual Studio.
  - If you use Visual Studio or VS Code IDE, the easiest way to add to Visual Studio or VS Code is to search and add a package or download from the TypeScript website.
  - Also, you can download TypeScript Tools for Visual Studio.

Initialize a TypeScript Project with tsconfig.json

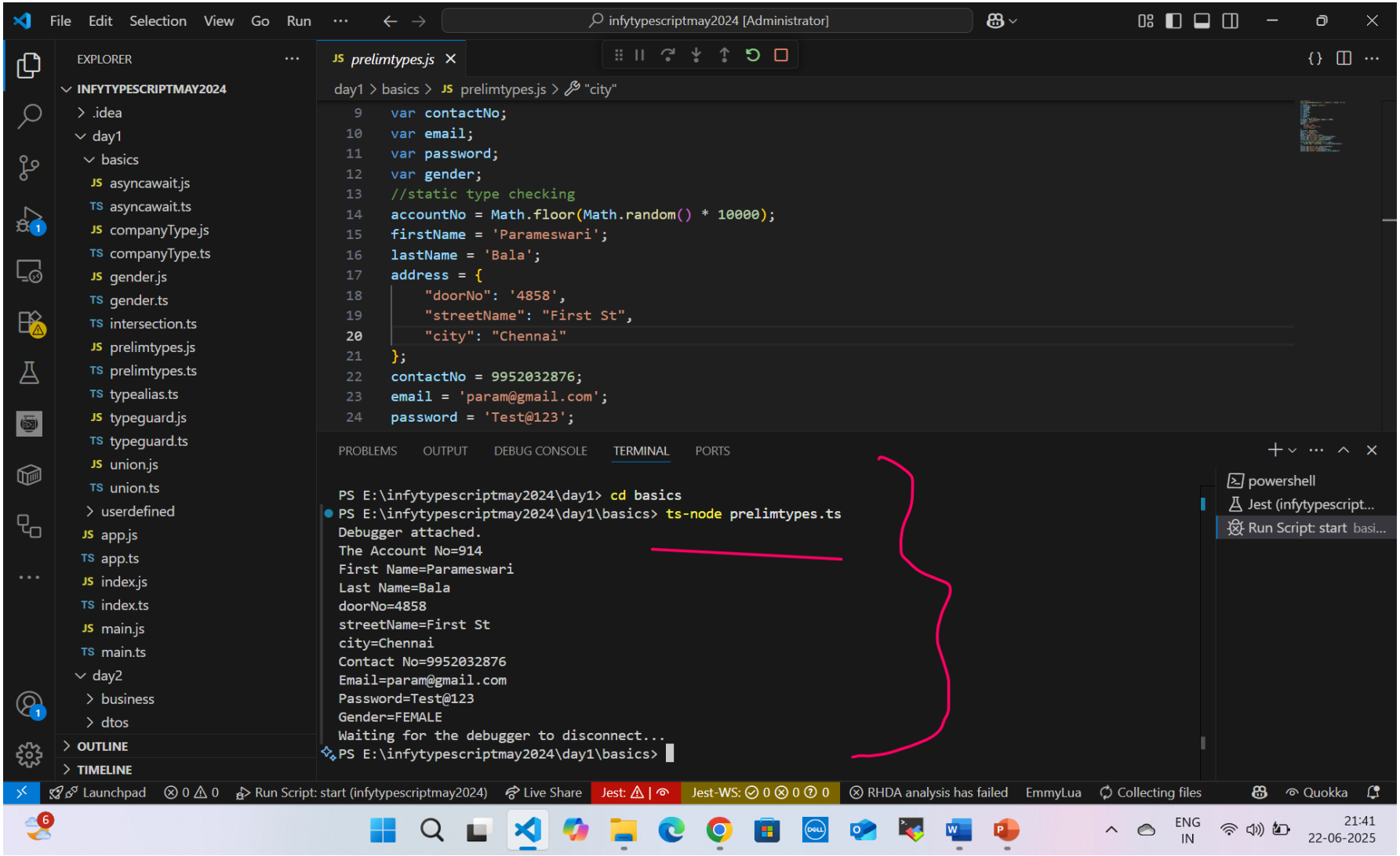
- Open VS Code terminal and run
- tsc --init
- This creates a tsconfig.json file with many options.

```
{
  "compilerOptions": {
    "target": "ES6",
    "module": "commonjs",
    "outDir": "./dist",
    "rootDir": "./src",
    "strict": true,
    "esModuleInterop": true
  }
}
```

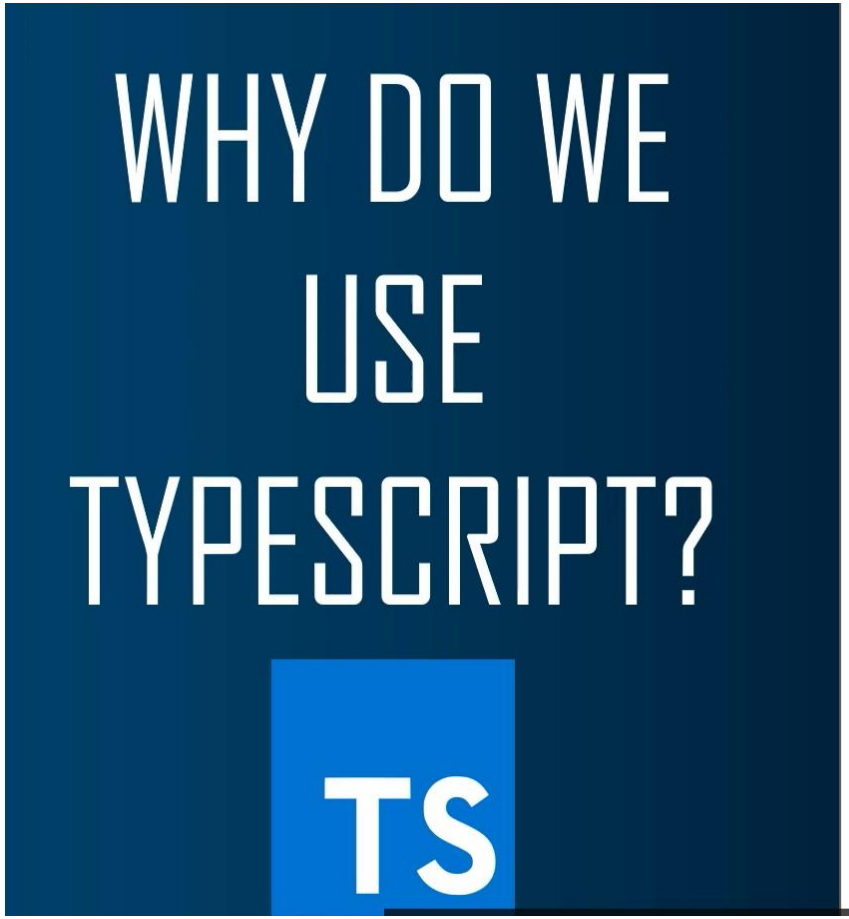
Install ts-node to Run Directly

- npm install -g ts-node
- ts-node src/index.ts

How to install TypeScript



Why do we useTypeScript



01 Using new features of ECMAScript

02 Static Typing

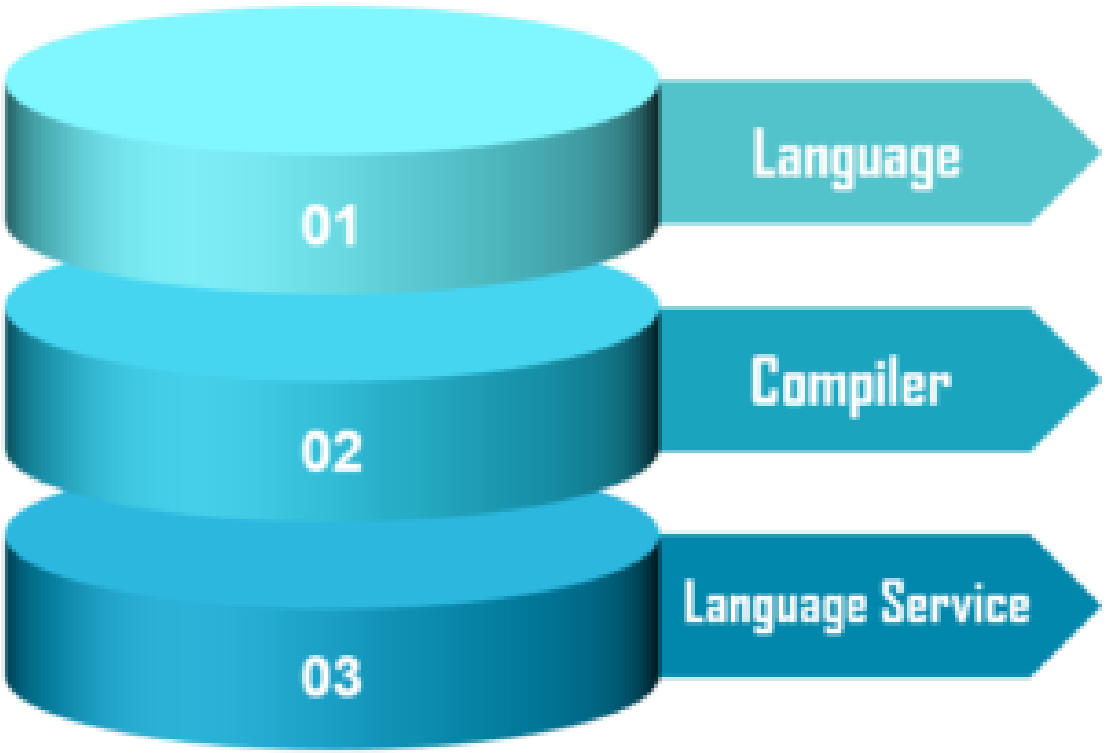
03 Type Inference

04 Better IDE Support

05 Strict Null Checking

06 Interoperability

Components of TypeScript



Components of TypeScript

- Language − It comprises of the syntax, keywords, and type annotations.

- The TypeScript Compiler − This compiler (tsc) converts the instructions written in TypeScript to its JavaScript equivalent.

- The TypeScript Language Service − The Language Service exposes an additional layer around the core compiler pipeline, editor-like applications.
- The language service supports the common set of typical editor operations.

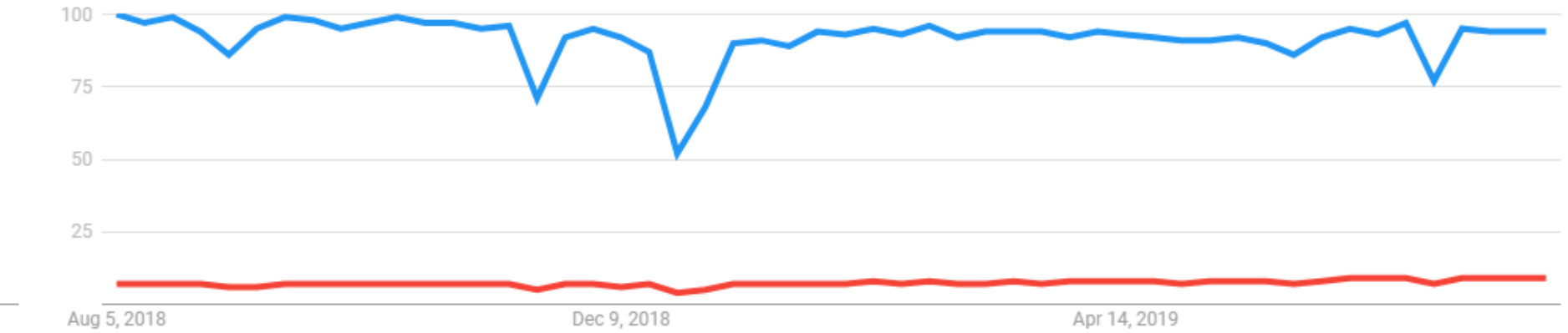
TypeScript vs JavaScript

	JavaScript	TypeScript
Language	Scripting language	Object-oriented programming language
Learning Curve	Flexible and easy to learn	a programmer should have prior scripting knowledge
Type	Lightweight, interpreted programming language	Strongly type object-oriented programming language
Client/Server side	Both client and server-side	Specially used in client-side
File Extension	.js	. ts or .tsx
Time	Faster	Takes time to compile the code
Data Binding	No concept of types and interfaces available	Concepts like types and interfaces used to describe the data being used.
Annotations	Annotations not required	Code must be annotated constantly to get the most out of TypeScript Features.

TypeScript vs JavaScript

Syntax	All the statements are written within the Script tag. The browser program starts interpreting all the text between these tags as a script<script>// javascript code</script>	A TypeScript program is composed of: Modules Functions Variables Statements Expressions Comments
Static Typing	There is no concept of Static typing in JavaScript	Supports static typing.
Support for Modules	Does not support modules	Gives support for modules
Interface	Does not have an interface	Has an interface
Optional parameter function	Does not support	Supports
Prototyping Feature	Does not have any such feature	Has a feature of prototyping
Community of developers	As JavaScript occupies the major chunk of codes, it is widely accepted and used by the programming community	TypeScript is new and has a relatively smaller community base.
Preference to choose	JavaScript is preferable to use in small coding projects.	TypeScript is an object-oriented language which makes the code more consistent, clean, simple and reusable. So it is better to use TypeScript for large projects.

TypeScript Trends



Typescript

TypeScript Cheat Sheet

Setup

Install TS globally on your machine

```
$ npm i -g typescript
```

Check version

```
$ tsc -v
```

Create the tsconfig.json file

```
$ tsc --init
```

Set the root (to compile TS files from) and output (for the compiled JS files) directories in tsconfig.json

```
"rootDir": "./src",
"outDir": "./public",
```

Compiling

Compile a specified TS file into a JS file of the same name, into the same directory (i.e. index.ts to index.js).

```
$ tsc index.ts
```

Tell tsc to compile specified file whenever a change is saved by adding the watch flag (-w)

```
$ tsc index.ts -w
```

Compile specified file into specified output file

```
$ tsc index.ts --outfile out/script.js
```

If no file is specified, tsc will compile all TS files in the "rootDir" and output in the "outDir". Add -w to watch for changes.

```
$ tsc -w
```

Strict Mode

In tsconfig.json, it is recommended to set strict to true. One helpful feature of strict mode is No Implicit Any:

```
// Error: Parameter 'a' implicitly has an 'any' type.
function logName(a) {
  console.log(a.name);
}
```

By @DoableDanny

Primitive Types

There are 7 primitive types in JS: string, number, bigInt, boolean, undefined, null, symbol.

Explicit type annotation

```
let firstname: string = 'Danny'
```

If we assign a value (as above), we don't need to state the type - TS will infer it ("implicit type annotation")

```
let firstname = 'Danny'
```

Union Types

A variable that can be assigned more than one type

```
let age: number | string;
age = 26;
age = "26";
```

Dynamic Types

The any type basically reverts TS back to JS.

```
let age: any = 100;
age = true;
```

Literal Types

We can refer to specific strings & numbers in type positions

```
let direction: 'UP' | 'DOWN';
direction = 'UP';
```

Objects

Objects in TS must have all the correct properties & value types

```
let person: {
 name: string;
 isProgrammer: boolean;
};

person = {
 name: 'Danny',
 isProgrammer: true,
};

person.age = 26; // Error - no age prop on person object

person.isProgrammer = 'yes'; // Error - should be boolean
```

Arrays

We can define what kind of data an array can contain

```
let ids: number[] = [];
ids.push(1);
ids.push("2"); // Error
```

Use a union type for arrays with multiple types

```
let options: (string | number)[];
options = [10, 'UP'];
```

If a value is assigned, TS will infer the types in the array.

```
let person = ['Delia', 48];
person[0] = true; // Error - only strings or numbers allowed
```

Tuples

A tuple is a special type of array with fixed size & known data types at each index. They're stricter than regular arrays.

```
let options: [string, number];
options = ['UP', 10];
```

Functions

We can define the types of the arguments, and the return type. Below, : string could be omitted because TS would infer the return type.

```
function circle(diam: number): string {
 return 'Circumf = ' + Math.PI * diam;
}
```

The same function as an ES6 arrow

```
const circle = (diam: number): string => 'Circumf = ' + Math.PI * diam;
```

If we want to declare a function, but not define it, use a function signature

```
let sayHi: (name: string) => void;

sayHi = (name: string) =>
 console.log('Hi ' + name);

sayHi('Danny'); // Hi Danny
```

Type Aliases

Allow you to create a new name for an existing type. They can help to reduce code duplication. They're similar to interfaces, but can also describe primitive types.

```
type StringOrNum = string | number;
let id: StringOrNum = 24;
```

Interfaces

Interfaces are used to describe objects. Interfaces can always be reopened & extended, unlike Type Aliases. Notice that `name` is `readonly`

```
interface Person {
 name: string;
 isProgrammer: boolean;
}

let p1: Person = {
 name: 'Delia',
 isProgrammer: false,
};

p1.name = 'Del'; // Error - read only
```

Two ways to describe a function in an interface

```
interface Speech {
 sayHi(name: string): string;
 sayBye: (name: string) => string;
}

let speech: Speech = {
 sayHi: function (name: string) {
  return 'Hi ' + name;
 },
 sayBye: (name: string) => 'Bye ' + name,
};
```

Extending an interface

```
interface Animal {
 name: string;
}

interface Dog extends Animal {
 breed: string;
}
```

The DOM & Type Casting

TS doesn't have access to the DOM, so use the non-null operator, !, to tell TS the expression isn't null or undefined

```
const link = document.querySelector('a')!;
```

If an element is selected by id or class, we need to tell TS what type of element it is via Type Casting

```
const form = document.getElementById('signup-form') as HTMLFormElement;
```

Generics

Generics allow for type safety in components where the arguments & return types are unkown ahead of time.

```
interface HasLength {
 length: number;
}

// logLength accepts all types with a length property
const logLength = <T extends HasLength>(a: T) => {
 console.log(a.length);
};

// TS "captures" the type implicitly
logLength('Hello'); // 5

// Can also explicitly pass the type to T
logLength<number[]>([1, 2, 3]); // 3
```

Declare a type, T, which can change in your interface.

```
interface Dog<T> {
 breed: string;
 treats: T;
}

// We have to pass in a type argument
let labrador: Dog<string> = {
 breed: 'labrador',
 treats: 'chew sticks, tripe',
};

let scottieDog: Dog<string[]> = {
 breed: 'scottish terrier',
 treats: ['turkey', 'haggis'],
};
```

Enums

A set of related values, as a set of descriptive constants

```
enum ResourceType {
 BOOK,
 FILE,
 FILM,
}
ResourceType.BOOK; // 0
ResourceType.FILE; // 1
```

Narrowing

Occurs when a variable moves from a less precise type to a more precise type

```
let age = getUserAge();
age // string | number

if (typeof age === 'string') {
 age; // string
}
```

Bit of History

Version number	Release date	Significant changes
0.8	October 1, 2012	
0.9	June 18, 2013	
1.1	October 6, 2014	performance improvements
1.3	November 12, 2014	protected modifier, tuple types
1.4	January 20, 2015	union types, let and const declarations, template strings, type guards, type aliases
1.5	July 20, 2015	ES6 modules, namespace keyword, for..of support, decorators
1.6	September 16, 2015	JSX support, intersection types, local type declarations, abstract classes and methods, user-defined type guard functions

Bit of History

1.7	November 30, 2015	async and await support,
1.8	February 22, 2016	constraints generics, control flow analysis errors, string literal types, allowJs
2.0	September 22, 2016	null- and undefined-aware types, control flow based type analysis, discriminated union types, never type, readonly keyword, type of this for functions
2.1	November 8, 2016	keyof and lookup types, mapped types, object spread and rest,
2.2	February 22, 2017	mix-in classes, object type,
2.3	April 27, 2017	async iteration, generic parameter defaults, strict option
2.4	June 27, 2017	dynamic import expressions, string enums, improved inference for generics, strict contravariance for callback parameters
2.5	August 31, 2017	optional catch clause variables

Bit of History

2.6	October 31, 2017	strict function types
2.7	January 31, 2018	constant-named properties, fixed length tuples
2.8	March 27, 2018	conditional types, improved keyof with intersection types
2.9	May 14, 2018	support for symbols and numeric literals in keyof and mapped object types
3.0	July 30, 2018	project references, extracting and spreading parameter lists with tuples
3.1	September 27, 2018	mappable tuple and array types
3.2	November 30, 2018	stricter checking for bind, call, and apply

Bit of History

3.3	31 January 2019	relaxed rules on methods of union types, incremental builds for composite projects
3.4	29 March 2019	faster incremental builds, type inference from generic functions, `readonly` modifier for arrays, `const` assertions, type-checking global `this`
3.5	29 May 2019	faster incremental builds, omit helper type, improved excess property checks in union types, smarter union type checking
3.6	28 August 2019	Stricter generators, more accurate array spread, better Unicode support for identifiers
3.7	5 November 2019	Optional chaining, nullish coalescing
3.8	20 February 2020	Type-only imports and exports, ECMAScript private fields, top-level `await`
3.9	12 May 2020	Improvements in inference, speed improvements
4.0	20 August 2020	Variadic tuple types, labeled tuple elements
4.1	19 November 2020	Template literal types, key remapping in mapped types, recursive conditional types
4.2	25 February 2021	Smarter type alias preservation, leading/middle rest elements in tuple types, stricter checks for the `in` operator, `abstract` construct signatures
4.3	26 May 2021	Separate write types on properties, `override` and the `--noImplicitOverride` flag, template string type improvements
4.4	26 August 2021	Control flow analysis of aliased conditions and discriminants, symbol and template string pattern index signatures

Bit of History

4.5	17 November 2021	Type and promise improvements, supporting lib from `node_modules`, template string types as discriminants, and `es2022` module
4.6	28 February 2022	Type inference and checks improvements, support for ES2022 target, better ECMAScript handling
4.7	24 May 2022	Support for ES modules, instantiation expressions, variance annotations for type parameters, better control-flow checks and type check improvements
4.8	25 August 2022	Intersection and union types improvements, better type inference
4.9	15 November 2022	`satisfies` operator, auto-accessors in classes (proposal), improvements in type narrowing and checks
5.0	16 March 2023	ES decorators (proposal), type inference improvements, `bundler` module resolution mode, speed and size optimizations
5.1	1 June 2023	Easier implicit returns for `undefined` and unrelated types for getters and setters
5.2	24 August 2023	`using` declarations and explicit resource management, decorator metadata and named and anonymous tuple elements
5.3	20 November 2023	Improved type narrowing, correctness checks and performance optimizations
5.4	6 March 2024	`Object.groupBy` and `Map.groupBy` support
5.5 (Beta)	25 April 2024	Inferred Type Predicates, Regular Expression Syntax Checking, and Type Imports in JSDoc

Declaring Variables

✅ 1. `let` — Block-scoped, mutable

```ts
let age: number = 25;
age = 30; // ✅ OK
```

- Scope: block (e.g., inside `{}`).
- Reassignment: allowed.
- Good for variables that may change later.

Declaring Variables

✅ 2. `const` — Block-scoped, immutable

```ts
const name: string = "Alice";
// name = "Bob"; ❌ Error: Cannot reassign a constant.
```

- Scope: block.
- Reassignment: **not allowed**.
- Good for constants or values that should not change.

Declaring Variables

✅ 3. `var` — Function-scoped, legacy

```ts
var count: number = 10;
```

- Scope: function-level.
- Reassignment: allowed.
- Avoid using `var` in modern TypeScript; use `let` or `const` instead.

TypeScript Basic Types

- Type Annotation – learn how to use type annotation to define the static type for variables, function parameters, and return values.
- Number – learn about the numeric types including floating-point numbers and big integers.
- String – show you how to use the string type in TypeScript.
- Boolean – guide you on the boolean type and how to use it effectively.
- Object Type – introduce you to the object type that represents non-primitive values.
- Array – show you how to use an array and its operations.
- Tuple – learn how to store a fixed number of elements with known types in a tuple.
- Enum – show you how to define a group of named constants by using the enum type.
- Any Type – learn how to use the `any` type to store a value of any type in a variable.
- Void type – show you how to use the void type as the return type of functions that do not return any value.

TypeScript Basic Types

- Never Type – learn how to use the `never` type that contains no value.
- Union Types – guide you on how to store a value of one or several types in a variable with the union type.
- Type Aliases – show you how to define new names for types using type aliases.
- String Literal Types – learn how to define a type that accepts only a specified string literal.
- Type Inference – explain where and how TypeScript infers types of variables.

What is Type Annotation in TypeScript

- TypeScript uses type annotations to explicitly specify types for identifiers such as variables, functions, objects, etc.
- TypeScript uses the syntax : type after an identifier as the type annotation, which type can be any valid type.
- Once an identifier is annotated with a type, it can be used as that type only.
- If the identifier is used as a different type, the TypeScript compiler will issue an error.

What is Type Annotation in TypeScript

- TypeScript uses type annotations to explicitly specify types for identifiers such as variables, functions, objects, etc.
- TypeScript uses the syntax : type after an identifier as the type annotation, which type can be any valid type.
- Once an identifier is annotated with a type, it can be used as that type only.
- If the identifier is used as a different type, the TypeScript compiler will issue an error.

Type annotations in variables and constants

The following syntax shows how to specify type annotations for variables and constants:

```
let variableName: type;
let variableName: type = value;
const constantName: type = value;
```

In this syntax, the type annotation comes after the variable or constant name and is preceded by a colon (`:`).

The following example uses `number` annotation for a variable:

```
let counter: number;
```

After this, you can only assign a number to the `counter` variable:

```
counter = 1;
```

Type annotations in variables and constants

If you assign a string to the `counter` variable, you'll get an error:

```
let counter: number;
counter = 'Hello'; // compile error
```

Error:

```
Type '"Hello"' is not assignable to type 'number'.
```

You can both use a type annotation for a variable and initialize it in a single statement like this:

```
let counter: number = 1;
```

Type annotations in variables and constants

In this example, we use the number annotation for the `counter` variable and initialize it to one.

The following shows other examples of primitive type annotations:

```
let name: string = 'John';
let age: number = 25;
let active: boolean = true;
```

In this example, the `name` variable gets the `string` type, the `age` variable gets the `number` type, and the `active` variable gets the `boolean` type.

Type Inference

- Four ways of variable declaration -
  1. Declare its type and value (as a literal) in one statement.
  2. Declare its type but no value. The value will be set to undefined.
  3. Declare its value but no type. The variable will be of type Any (that is, an old-school dynamic JavaScript variable), but its type may be inferred based on its value.
  4. Declare neither value nor type. The variable will be of type Any, and its value will

```
// Option 1: Declare variable type and value
var message1:string = "hello from TS";

// Option 2: Declare variable type, but no value
var message2:string;
// value could be assigned later
message2 = "hello from TS";

// Option 3: Declare the value but no type
// TypeScript would try to infer the value
var message3 = "hello from TS";

// Option 4: Neither declare type or value
// Type is inferred as Any - value of any type can be assigned
var message4;
```

Array


```
var cities:string[] = ["Berlin","Bangalore","New York"]
var primes:number[] = [1,3,5,7,11,13]
var bools:boolean[] = [true,false,false,true]
```

```
1  function displayMessage(name:string, message:string, salutation:string='Mr',
2  ...otherInfo:string[]){
3      var addlMessage:string;
4      addlMessage = salutation + ' ' + name + "\n"+ message + ' '
5      + '\nAddl message: ';
6
7      for (var index in otherInfo){
8          addlMessage += otherInfo[index] + ', ';
9      }
10     alert(addlMessage);
11 }
12
13 displayMessage('Aniruddha', 'hello');
14 displayMessage('XYZ', 'hello', 'dr','param1','param2','param3','param4');
```

Objects

Objects

To specify a type for an object, you use the object type annotation. For example:

```
let person: {
   name: string;
   age: number
};

person = {
   name: 'John',
   age: 25
}; // valid
```

In this example, the `person` object only accepts an object that has two properties: `name` with the `string` type and `age` with the `number` type.

Enum

- Addition to JavaScript datatypes. Similar to C# enum
- Like languages like C#, an enum is a way of giving more friendly names to sets of numeric values.
- By default, enums begin numbering their members starting at 0. You can change this by manually setting the value of one its members.

```
enum Color{
    Red,Green,Blue
}

var blue:Color = Color.Blue
var red:Color = Color.Red
```

```
The property 'green' does not exist on value of type 'typeof Color'.
any
var green:Color = Color.green
```

```
enum Color{
    Red,Green,Blue
}

enum Color2{
    Red=0,Green=1,Blue=2
}
```

```
enum Color{
    Red,Green,Blue
}

enum Color2{
    Red=1,Green=3,Blue=5
}
```

Any

- Useful to describe the type of variables that we may not know when we are writing the application.
- May come from dynamic content, eg from the user or 3rd party library.
- Allows to opt-out of type-checking and let the values pass through compile-time checks.
- Same as not declaring any datatype – uses JavaScript's dynamic nature

```
var notSure: any


var list:any[] = [1, true, "free"]
list[1] = 100
```

Void

- Perhaps the opposite in some ways to 'any' is 'void',
- the absence of having any type at all.
- Commonly used as the return type of functions that do not return a value

```
function warnUser(): void {
    alert("This is my warning message");
}
```

Control Flow Statements

- if...else – learn how to execute code based on a condition.
- switch..case – show you how to use the switch statement to allow a number of possible execution paths.
- for – create a loop that executes a specified number of times.
- while – create a pretest loop that executes as long as a condition is true.
- do...while – learn how to create a posttest loop that executes until a condition is false.
- break – show you how to use the break statement to terminate a loop or a switch.
- continue – learn how to skip to the end of a loop and continue to the next iteration.

TypeScript If elseif else

When you want to execute code based on multiple conditions, you can use the `if...else if...else` statement.

The if...else if...else statement can have one or more `else if` branches but only one `else` branch.

For example:

```
let discount: number;
let itemCount = 11;

if (itemCount > 0 && itemCount <= 5) {
    discount = 5;  // 5% discount
} else if (itemCount > 5 && itemCount <= 10) {
    discount = 10; // 10% discount
} else {
    discount = 15; // 15%
}

console.log(`You got ${discount}% discount. `)
```

TypeScript Switch Case

The following shows the syntax of the `switch...case` statement:

```
switch ( expression ) {
   case value1:
       // statement 1
       break;
   case value2:
       // statement 2
       break;
   case valueN:
       // statement N
       break;
   default:
       //
       break;
}
```

Non-Nullable Types in TypeScript

- In TypeScript, non-nullable types are types that cannot be null or undefined, unless explicitly allowed.
- By default, in strict mode (strictNullChecks: true in tsconfig.json), TypeScript treats null and undefined as distinct types.

```
let name: string = "Alice";
name = null;        // ❌ Error
name = undefined;   // ❌ Error
```

- This ensures safer code, preventing runtime errors caused by null or undefined values.

Non-Nullable Types in TypeScript

- How to Allow Null or Undefined (Nullable Types)
- You must explicitly include null or undefined in the type:

```
let middleName: string | null = null;          // ✅ OK
let status: string | undefined = undefined;    // ✅ OK
```

Utility Types

- In TypeScript, utility types are built-in types that help transform or construct new types based on existing ones.
- They make your code cleaner, more expressive, and reduce duplication.

Mapped Type Utilities

Utility	Description
Partial<T>	Makes all properties of T optional.
Required<T>	Makes all properties of T required.
Readonly<T>	Makes all properties of T read-only.
Record<K, T>	Constructs an object type with keys K and values of type T.
Pick<T, K>	Creates a new type by picking a subset of properties from T.
Omit<T, K>	Creates a new type by omitting properties K from T.

Type Transform Utilities

Utility	Description
Exclude<T, U>	Excludes from T those types that are assignable to U.
Extract<T, U>	Extracts from T those types that are assignable to U.
NonNullable<T>	Removes null and undefined from T.

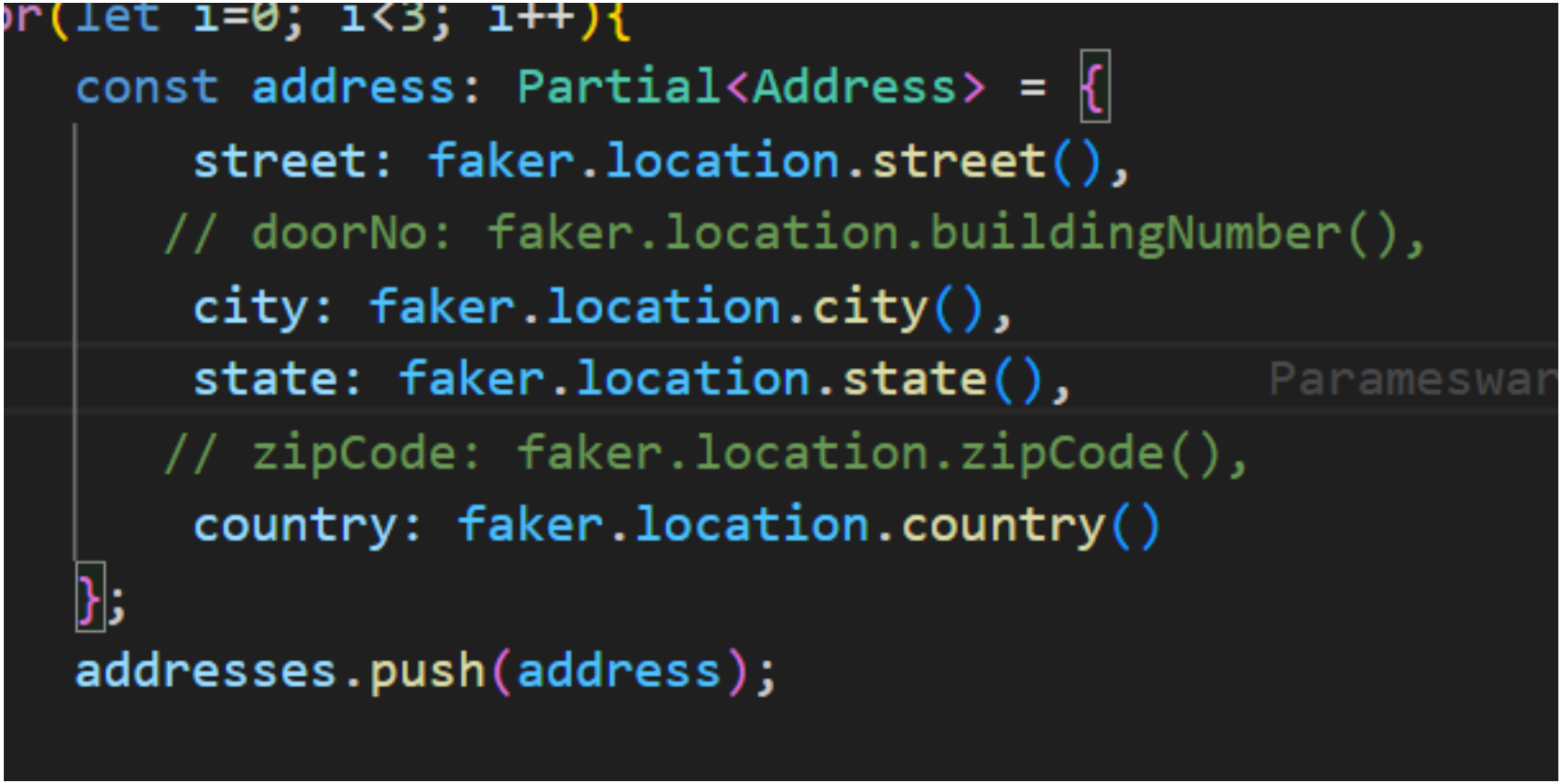
Function Type Utilities

Utility	Description
ReturnType<T>	Gets the return type of a function type T.
Parameters<T>	Gets the parameter types of a function type T.
ConstructorParameters<T>	Gets the types of a constructor function's parameters.
InstanceType<T>	Gets the instance type of a class constructor.

Advanced / Others

Utility	Description
ThisParameterType<T>	Extracts the type of the this parameter in a function.
OmitThisParameter<T>	Removes the this parameter from a function type.
Awaited<T>	Gets the resolved type of a Promise or async function.

Partial Type



Required Type

- It removes optional (?) modifiers from all properties of T.
- type User = {
- name?: string;
- age?: number;
- };

- type RequiredUser = Required<User>;
- It becomes mandatory and removes optional

Read Only

- The Readonly<T> utility type makes all properties of an object type T immutable, meaning they cannot be reassigned after initial assignment.
- type User = {
- name: string;
- age: number;
- };
- const user: Readonly<User> = {
- name: "Alice",
- age: 30
- };
- user.age = 31; // ❌ Error: Cannot assign to 'age' because it is a read-only property.

Record<K, T> in TypeScript

- The Record<K, T> utility type constructs an object type with a set of keys K and value type T.
- Record<K, T>
  - K: a union of property keys (string, number, or symbol).
  - T: the type of values associated with each key.

Record<K, T> in TypeScript

- type Role = 'admin' | 'user' | 'guest';

- type RolePermissions = Record<Role, boolean>;

- const permissions: RolePermissions = {
- admin: true,
- user: true,
- guest: false,
- };

Record<K, T> in TypeScript


```
type Product = {
  name: string;
  price: number;
};

type ProductMap = Record<string, Product>;

const products: ProductMap = {
  apple: { name: 'Apple', price: 10 },
  banana: { name: 'Banana', price: 5 }
};
```

Pick

- Pick<T, K>
- T: the original object type.
- K: a union of keys from T that you want to "pick".

Pick

- type User = {
- id: number;
- name: string;
- email: string;
- isActive: boolean;
- };
- type PublicProfile = Pick<User, 'id' | 'name'>;
- Now Public Profile becomes:
- type PublicProfile = {
- id: number;
- name: string;
- };

Omit

- Omit<T, K>
- T: the original object type.

- K: a union of keys to exclude (must be keys of T).

Omit

- type User = {
- id: number;
- name: string;
- email: string;
- isActive: boolean;
- };
- type PrivateUser = Omit<User, 'email' | 'isActive'>;
- Now PrivateUser becomes:
- type PrivateUser = {
- id: number;
- name: string;
- };

Exclude<T, U>

- The Exclude<T, U> utility type constructs a new type by excluding from union type T all members that are assignable to type U.
- type Roles = 'admin' | 'user' | 'guest';
- type LimitedRoles = Exclude<Roles, 'guest’>;
- type Primitive = string | number | boolean | null | undefined;
- type NonNullablePrimitive = Exclude<Primitive, null | undefined>;

Extract

- The Extract<T, U> utility type constructs a new type by selecting from T only those members that are assignable to U.
- type Roles = 'admin' | 'user' | 'guest';

- type StaffRoles = Extract<Roles, 'admin' | 'user'>;
- // Result: 'admin' | 'user'

Non Nullable Utility Type

- TypeScript provides a built-in utility type: NonNullable<T>

ts

```ts
type T = string | null | undefined;
type TNonNull = NonNullable<T>; // → string
```

ts

```ts
function greet(name: NonNullable<string | null | undefined>) {
  console.log("Hello, " + name);
}

greet("Bob");      // ✅ OK
greet(null);       // ❌ Error
greet(undefined);  // ❌ Error
```

Return Type

- The ReturnType<T> utility type extracts the return type of a function type T.
- function getUser() {
- return {
- id: 1,
- name: "Alice",
- isActive: true
- };
- }

- type UserReturn = ReturnType<typeof getUser>;

Return Type

- Now UserReturn becomes:
- type UserReturn = {
- id: number;
- name: string;
- isActive: boolean;};

Parameters

- It **returns a tuple** representing the types of the function's parameters.
- function createUser(name: string, age: number, isActive: boolean) {
- return { name, age, isActive };
- }
- type Params = Parameters<typeof createUser>;
- type Params = [string, number, boolean];

Function

Function Overview

- Functions are the fundamental building block of any applications in JavaScript.
- JavaScript is a functional programming language, and so supports first class functions.
- Allows build up layers of abstraction, mimicking classes, information hiding, and modules (JavaScript does not support class, module, private members).
- In TypeScript, while there are classes and modules, function still play the key role in describing how to 'do' things.

Function Overview

- TypeScript adds some new capabilities to the standard JavaScript functions to make them easier to work with.
- Type Annotation for parameter and return type
- Optional and Default Parameter
- Rest Parameter
- Function Overloads

Function Arguments and Return Types

The following shows a function annotation with parameter type annotation and return type annotation:

```
let greeting : (name: string) => string;
```

In this example, you can assign any function that accepts a string and returns a string to the `greeting` variable:

```
greeting = function (name: string) {
    return `Hi ${name}`;
};
```

The following causes an error because the function that is assigned to the `greeting` variable doesn't match its function type.

```
greeting = function () {
    console.log('Hello');
};
```

Error:

```
Type '() => void' is not assignable to type '(name: string) => string'. Type 'void' is not assignable to type 'string'.
```

Function Overloads


```
function add(var1:string, var2:string):string;
function add(var1:number, var2:number):number;

function add(var1, var2):any{
    if (typeof var1 == "number" && typeof var2 == "number"){
        return var1 + var2;
    }
    if (typeof var1 == "string" && typeof var2 == "string"){
        return var1 + ' ' + var2;
    }
}

var r1 = add(10,20);
var r2 = add('hello','world');

alert(r1); // 30
alert(r2); // hello world
```

Function Overloads (2)

```
1  class Point{
2      constructor(public x:number, public y:number){
3      }
4  }
5
6  function add(var1:string, var2:string):string;
7  function add(var1:number, var2:number):number;
8  function add(var1:Point, var2:Point):Point;
9  function add(var1, var2):any{
10     if (typeof var1 == "number" && typeof var2 == "number"){
11         return var1 + var2;
12     }
13     if (typeof var1 == "string" && typeof var2 == "string"){
14         return var1 + ' ' + var2;
15     }
16     if (typeof var1 == "object" && typeof var2 == "object"){
17         var newPoint:Point = new Point(var1.x + var2.x, var1.y + var2.y);
18         return newPoint;
19     }
20 }
21
22 var r1 = add(10,20);
23 var r2 = add('hello','world');
24
25 var p1:Point = new Point(10,20);
26 var p2:Point = new Point(40,60);
27 var p3:Point = add(p1,p2);
28
29 alert(r1); // 30
30 alert(r2); // hello world
31
32 alert(p3.x + ',' + p3.y); // 50,80
```

Optional & Default Parameter

- In JavaScript, you can call a function without passing any arguments even though the function specifies parameters.
- Therefore, JaveScript supports the optional parameters by default.
- In TypeScript, the compiler checks every function call and issues an error in the following cases:
- The number of arguments is different from the number of parameters specified in the function.
- Or the types of arguments are not compatible with the types of function parameters.
- Because the compiler thoroughly checks the passing arguments, you need to annotate optional parameters to instruct the compiler not to issue an error when you omit the arguments.

Optional & Default Parameter

To make a function parameter optional, you use the `?` after the parameter name. For example:

```
function multiply(a: number, b: number, c?: number): number {

    if (typeof c !== 'undefined') {
        return a * b * c;
    }
    return a * b;
}
```

How it works:

- First, use the `?` after the `c` parameter.
- Second, check if the argument is passed to the function by using the expression `typeof c !== 'undefined'` .

> Note that if you use the expression `if(c)` to check if an argument is not initialized, you would find that the empty string or zero would be treated as `undefined` .

Optional & Default Parameter

JavaScript supported default parameters since ES2015 (or ES6) with the following syntax:

```
function name(parameter1=defaultValue1,...) {
   // do something
}
```

In this syntax, if you don't pass arguments or pass the `undefined` into the function when calling it, the function will take the default initialized values for the omitted parameters. For example:

```
function applyDiscount(price, discount = 0.05) {
    return price * (1 - discount);
}

console.log(applyDiscount(100)); // 95
```

In this example, the `applyDiscount()` function has the discount parameter as a default parameter.

When you don't pass the `discount` argument into the `applyDiscount()` function, the function uses a default value which is `0.05`.

Optional & Default Parameter

Similar to JavaScript, you can use default parameters in TypeScript with the same syntax:

```
function name(parameter1:type=defaultvalue1, parameter2:type=defaultvalue2,...) {
   //
}
```

The following example uses default parameters for the `applyDiscount()` function:

```
function applyDiscount(price: number, discount: number = 0.05): number {
    return price * (1 - discount);
}

console.log(applyDiscount(100)); // 95
```

Notice that you cannot include default parameters in function type definitions. The following code will result in an error:

```
let promotion: (price: number, discount: number = 0.05) => number;
```

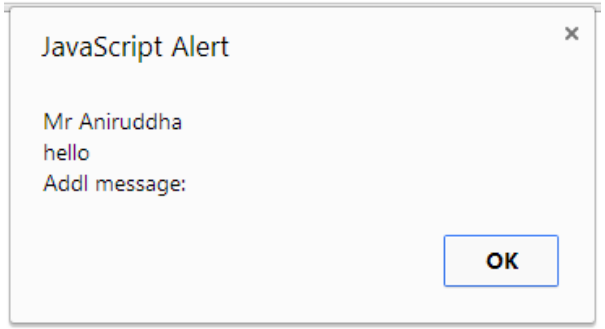
Error:

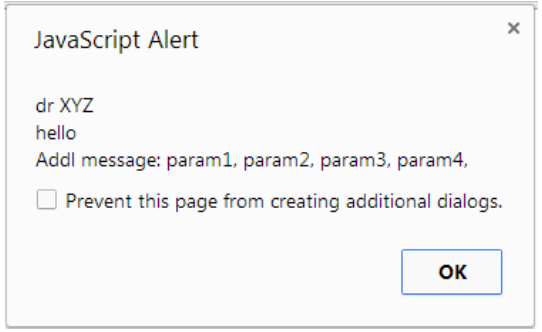
```
error TS2371: A parameter initializer is only allowed in a function or constructor implementation.
```

Rest Parameter

- Declared as ... `paramName:[paramType]`

```
1  function displayMessage(name:string, message:string, salutation:string='Mr',
2  ...otherInfo:string[]){
3      var addlMessage:string;
4      addlMessage = salutation + ' ' + name + "\n"+ message + ' '
5      + '\nAddl message: ';
6
7      for (var index in otherInfo){
8          addlMessage += otherInfo[index] + ', ';
9      }
10     alert(addlMessage);
11 }
12
13 displayMessage('Aniruddha', 'hello');
14 displayMessage('XYZ', 'hello', 'dr','param1','param2','param3','param4');
```





TypeScript Rest Parameters

A rest parameter allows a function to accept zero or more arguments of the specified type. In TypeScript, the rest parameters follow these rules:

- A function has only one rest parameter.
- The rest parameter appears last in the parameter list.
- The type of the rest parameter is an array type.

To declare a rest parameter, you prefix the parameter name with three dots and use the array type as the type annotation:

```
function fn(...rest: type[]) {
   //...
}
```

The following example shows how to use the rest parameter:

```
function getTotal(...numbers: number[]): number {
    let total = 0;
    numbers.forEach((num) => total += num);
    return total;
}
```

In this example, the `getTotal()` calculates the total of numbers passed into it.

TypeScript Rest Parameters

Since the numbers parameter is a rest parameter, you can pass one or more numbers to calculate the total:

```
console.log(getTotal()); // 0
console.log(getTotal(10, 20)); // 30
console.log(getTotal(10, 20, 30)); // 60
```

Usage in Function Parameters

- Without Non Nullable:

```ts
function getLength(str: string | null) {
  return str.length; // ❌ Error: Object is possibly 'null'
}
```

- With Check or Assertion:

```ts
function getLength(str: string | null) {
  if (str !== null) {
    return str.length; // ✅ Safe
  }
}
```

Usage in Function Parameters

- Or using the ! non-null assertion:

```ts
function getLength(str: string | null) {
  return str!.length; // ✅ Tells compiler: "I know it's not null"
}
```

- Or using the ! non-null assertion:

```ts
function getLength(str: string | null) {
  return str!.length; // ✅ Tells compiler: "I know it's not null"
}
```

Why Use Non-Nullable Types?

Reason	Benefit
Avoid null/undefined bugs	Reduces runtime crashes
Enforces safe programming	Compiler catches potential issues
Easier to reason about data	Especially in large codebases
Matches strict API contracts	Especially useful for backend/DB validations

Basic Data Structure

- Arrays
- Ordered collections of items (of any type).

```ts
let numbers: number[] = [1, 2, 3];
let names: string[] = ["Alice", "Bob"];
let mixed: (number | string)[] = [1, "two", 3];
```

- Built-in methods: push(), pop(), map(), filter(), reduce(), etc.

Basic Data Structure

- **Tuples**
- Fixed-length array with specified types for each element.

```ts
let user: [number, string] = [1, "Alice"];
```

- Use cases: Representing a data row or key-value pairs with fixed types.

Basic Data Structure

- Objects
- Collections of key-value pairs.

```ts
let person: { name: string; age: number } = {
  name: "Bob",
  age: 25,
};
```

- Can define reusable object structures using interface or type.

Basic Data Structure

- Sets
- Unordered collection of unique values.

```ts
let ids: Set<number> = new Set([1, 2, 3, 1]);
console.log(ids); // → Set { 1, 2, 3 }
```

- Fast lookup and uniqueness enforcement.

Basic Data Structure

- Maps
- Unordered collection of unique values.

```
let userMap: Map<number, string> = new Map();
userMap.set(1, "Alice");
userMap.set(2, "Bob");
```

- Useful when keys are not strings (e.g., objects or numbers).

Basic Data Structure

- Stacks
- LIFO (Last-In, First-Out) data structure.

```ts
let stack: number[] = [];
stack.push(10);
stack.push(20);
let top = stack.pop(); // 20
```

- Built using arrays.

Basic Data Structure

- Queues
- FIFO (First-In, First-Out) data structure.

```ts
let queue: number[] = [];
queue.push(1);  // enqueue
queue.push(2);
let first = queue.shift(); // dequeue → 1
```

- Built using arrays.

Basic Data Structure

- Linked Lists, Trees, Graphs
- These are not built-in but can be implemented manually.
- Example: Singly linked list node

```
class ListNode {
  value: number;
  next: ListNode | null;

  constructor(value: number) {
    this.value = value;
    this.next = null;
  }
}
```

Summary Table

Structure	Use Case	Built-in?
Array	Ordered data	✅
Tuple	Fixed-type group	✅
Object	Key-value pairs	✅
Set	Unique values	✅
Map	Key-value pairs (any key type)	✅
Stack	LIFO operations	✅ (via array)
Queue	FIFO operations	✅ (via array)
Linked List, Tree, Graph	Custom DS	❌ (manual)

Class

Class

- Properties and fields to store data
- Methods to define behavior
- Events to provide interactions between different objects and classes

Field and Property


```
class Person {
    ssn: string;
    firstName: string;
    lastName: string;

    constructor(ssn: string, firstName: string, lastName: string) {
        this.ssn = ssn;
        this.firstName = firstName;
        this.lastName = lastName;
    }

    getFullName(): string {
        return `${this.firstName} ${this.lastName}`;
    }
}
```

TypeScript Access Modifiers

- Access modifiers change the visibility of the properties and methods of a class. TypeScript provides three access modifiers:
  - private
  - protected
  - public
- Note that TypeScript controls the access logically during compilation time, not at runtime.

Private Access Modifiers

- The private modifier
  - The private modifier limits the visibility to the same class only.
  - When you add the private modifier to a property or method, you can access that property or method within the same class.
  - Any attempt to access private properties or methods outside the class will result in an error at compile time.
  - The following example shows how to use the private modifier to the snn, firstName, and lastName properties of the person class:

Private Access Modifiers


```
class Person {
    private ssn: string;
    private firstName: string;
    private lastName: string;
    // ...
}
```

Once the `private` property is in place, you can access the `ssn` property in the constructor or methods of the `Person` class. For example:

```
class Person {
    private ssn: string;
    private firstName: string;
    private lastName: string;

    constructor(ssn: string, firstName: string, lastName: string) {
        this.ssn = ssn;
        this.firstName = firstName;
        this.lastName = lastName;
    }

    getFullName(): string {
        return `${this.firstName} ${this.lastName}`;
    }
}
```

Public Access Modifiers

- The public modifier allows class properties and methods to be accessible from all locations.
- If you don’t specify any access modifier for properties and methods, they will take the public modifier by default.
- For example, the getFullName() method of the Person class has the public modifier.
- The following explicitly adds the public modifier to the getFullName() method:

Public Access Modifiers


```
class Person {
    // ...
    public getFullName(): string {
        return `${this.firstName} ${this.lastName}`;
    }
    // ...
}
```

The protected modifier

- The protected modifier allows properties and methods of a class to be accessible within the same class and within subclasses.
- When a class (child class) inherits from another class (parent class), it is a subclass of the parent class.
- The TypeScript compiler will issue an error if you attempt to access the protected properties or methods from anywhere else.

The protected modifier

To add the protected modifier to a property or a method, you use the `protected` keyword. For example:

```
class Person {

    protected ssn: string;

    // other code
}
```

To make the code shorter, TypeScript allows you to both declare properties and initialize them in the constructor like this:

```
class Person {
    constructor(protected ssn: string, private firstName: string, private lastName: string) {
        this.ssn = ssn;
        this.firstName = firstName;
        this.lastName = lastName;
    }

    getFullName(): string {
        return `${this.firstName} ${this.lastName}`;
    }
}
```

TypeScript readonly

- TypeScript provides the readonly modifier that allows you to mark the properties of a class immutable.
- The assignment to a readonly property can only occur in one of two places:
  - In the property declaration.
  - In the constructor of the same class.

TypeScript readonly


```
class Person {
    readonly birthDate: Date;

    constructor(birthDate: Date) {
        this.birthDate = birthDate;
    }
}
```

The following attempts to reassign the `birthDate` property that results in an error:

```
let person = new Person(new Date(1990, 12, 25));
person.birthDate = new Date(1991, 12, 25); // Compile error
```

TypeScript read only


```
class Person {
    constructor(readonly birthDate: Date) {
        this.birthDate = birthDate;
    }
}
```

Readonly vs. const

The following shows the differences between readonly and const:

	readonly	const
Use for	Class properties	Variables
Initialization	In the declaration or in the constructor of the same class	In the declaration

Constructor

- Uses constructor keyword
- public by default, can not be private

```
class Employee{
    private name:string
    private basic:number
    private allowance:number

    constructor(name:string, basic:number, allowance:number){
        this.name = name
        this.basic = basic
        this.allowance = allowance
    }
```

Constructor shortcut


```
class Employee{
    private name:string
    private basic:number
    private allowance:number

    constructor(name:string, basic:number, allowance:number){
        this.name = name
        this.basic = basic
        this.allowance = allowance
    }

    public getSalary():number{
        return this.basic + this.allowance
    }
}
```

```
1  class Employee{
2      constructor(public name:string, public basic: number,
3      public allowance:number){
4          // no initialization required
5      }
6
7      public getSalary(){
8          return this.basic + this.allowance;
9      }
10 }
11
12 var emp = new Employee('XYZ',100,10);
13 alert(emp.getSalary());
```

Events

```
class Employee{
    name:string // public field
    private _hiddenField:string // private field

    private _salary:number // private backing field

    // public property
    get Salary():number{
        return this._salary
    }
    set Salary(value:number){
        if (value <= 0){
            throw "salary can not be less than 0"
        }
        else{
            this._salary = value
        }
    }
}
```

```
var emp: Employee = new Employee()
try{
    emp.Salary = -100
}
catch(error){
    alert(error)
}
emp.Salary = 50
alert(emp.Salary)
```

```
var emp: Employee = new Employee()
try{
    emp.Salary = -100
}
catch(er
    aler
}
emp.Sala
alert(em
```

name string

Salary

Access Modifiers

- public (default) - member is available to all code in another module.
- private - member is available only to other code in the same assembly.

```
1  class Employee{
2      public public_method(){
3          alert('public_method called')
4      }
5      private private_method(){
6          alert('private_method called')
7      }
8  }
9
10 var emp: Employee = new Employee()
```

```
9
10 'Employee.private_method' is inaccessible.
11 (): void
12 emp.private_method();
13
14 emp.public_method();
15
```

public_method (): void

Static Methods

- TypeScript supports static members (methods)
- *static* methods are visible on the class itself rather than on the instances

```
class Employee{
    instanceMethod(){
        alert('Employee.instanceMethod called')
    }

    static staticMethod(){
        alert('Employee.staticMethod called')
    }
}

new Employee().instanceMethod();
Employee.staticMethod();
```

Class


```
1  class Employee{
2      private firstName:string
3      private lastName:string
4      private basic:number
5      private allowance:number
6      private tax:number
7
8      constructor(firstName:string,lastName:string, basic:number,allowance:number, tax:number){
9          this.firstName = firstName
10         this.lastName = lastName
11         this.basic = basic
12         this.allowance = allowance
13         this.tax = tax
14     }
15
16     getName(){
17         return this.firstName + " " + this.lastName
18     }
19
20     calculateSalary(){
21         return this.basic + this.allowance - this.tax
22     }
23 }
24
25 var emp: Employee = new Employee("Bill","Gates",1000,100,200)
26 alert(emp.calculateSalary())
27
```

JavaScript Constructor Pattern


```
// Constructor Pattern for Idiomatic JavaScript
var Employee = (function(){
        function Employee(name, basic, allowance){
            this.name = name
            this.basic = basic
            this.allowance = allowance
        }

        Employee.prototype.getSalary = function(){
            return this.basic + this.allowance
        }

        return Employee
})()


   var emp = new Employee("AC",100,20);
   alert(emp.name)
   alert(emp.getSalary())
```

JavaScript Constructor Pattern (2)

TypeScript

Select...

Share

```typescript
// Constructor Pattern for Idiomatic JavaScript
var Employee = (function(){
        function Employee(name, basic, allowance){
            this.name = name
            this.basic = basic
            this.allowance = allowance
        }

        Employee.prototype.getSalary = function(){
            return this.basic + this.allowance
        }

        return Employee
})()

var emp = new Employee("AC",100,20)
alert(emp.name)
alert(emp.getSalary())
```

Run

JavaScript

```javascript
// Constructor Pattern for Idiomatic JavaScript
var Employee = (function () {
    function Employee(name, basic, allowance) {
        this.name = name;
        this.basic = basic;
        this.allowance = allowance;
    }

    Employee.prototype.getSalary = function () {
        return this.basic + this.allowance;
    };

    return Employee;
})();

var emp = new Employee("AC", 100, 20);
alert(emp.name);
alert(emp.getSalary());
```

Class – TypeScript uses same Constructor Pattern

TypeScript | Select... | Share

```typescript
class Employee{
    private name:string
    private basic:number
    private allowance:number

    constructor(name:string, basic:number, allowance:number){
        this.name = name
        this.basic = basic
        this.allowance = allowance
    }

    public getSalary():number{
        return this.basic + this.allowance
    }
}

var emp = new Employee("Aniruddha",100,20)
alert(emp.getSalary())
```

Run | JavaScript

```javascript
var Employee = (function () {
    function Employee(name, basic, allowance) {
        this.name = name;
        this.basic = basic;
        this.allowance = allowance;
    }
    Employee.prototype.getSalary = function () {
        return this.basic + this.allowance;
    };
    return Employee;
})();

var emp = new Employee("Aniruddha", 100, 20);
alert(emp.getSalary());
```

Inheritance

- TypeScript supports inheritance of class through `extends` keyword

```
1  class Person{
2      constructor(public name:string, public age:number){
3      }
4      showInfo(){
5          alert("Name: " + this.name + " Age: " + this.age);
6      }
7  } // Base class
8
9  class Employee extends Person{
10     constructor(name, age, public salary:number){
11         super(name,age); // super calls the constructor of base class
12     }
13     showInfo(){
14         alert("Name:" + this.name + " Age:" + this.age + " Salary:" + this.salary);
15     }
16 } // Class that inhertits from Base class
17
18 var per:Person = new Person('Aniruddha',40);
19 per.showInfo(); // calls showInfo of Person class
20
21 var emp:Person = new Employee('Bill',55,100);
22 emp.showInfo(); // calls showInfo of Employee class
```

Module

Module

- Modules can be defines using `module` keyword
- A module can contain sub module, class, interface or enum. Can not directly contain functions (similar to C#, Java)
- Modules can be nested (sub module)
- Class, Interfaces can be exposed using `export` keyword

```
module Utils{
    export class Math{
        public add(x:number, y:number): number{
            return x+y
        }
    }
    // not accessible out side the module
    class Helper{
        help(){}
    }
}

// can not access the class without referencing Module
var m = new Utils.Math
m.add(20,30)
```

Interface

Interface

- Declared using `interface` keyword

```
1  interface Employee{
2      FirstName:string;
3      LastName:string
4  }
5
6  var emp:Employee = {FirstName:"Bill",LastName:"Gates"};
7  alert(emp.FirstName);
8
```

- Like many other TypeScript feature it's purely a Design time feature. No additional code is emitted for this!
- TS compiler shows error when Interface signature and implementation does not match

```
1  interface Employee{
2      FirstName:string;
3      LastName:string
4  }
5
6  Cannot convert '{ FirstName: string; }' to 'Employee': Type '{ FirstName: string;
7  }' is missing property 'LastName' from type 'Employee'.
8  string
9  var emp:Employee = {FirstName:"Bill"};
10 alert(emp.FirstName);
11
```

Interface (Cont'd)


```
// ShowEmployeeDetails expects an Employee object with FirstName, LastName and Age property
function ShowEmployeeDetails(emp){
    alert('hello ' + emp.FirstName + ' ' + emp.LastName + '. Your age is ' + emp.Age);
}

var emp1:Employee = {FirstName:'Bill',LastName:'Gates',Age:50};
var emp2 = 'is this an employee?'
var emp3 = {FirstName:'Bill',Age:50};

ShowEmployeeDetails(emp1); // Works as expected
ShowEmployeeDetails(emp2); // not an Employee object, shows undefined
ShowEmployeeDetails(emp3); // does not have LastName, shows undefined
```

```
interface Employee{
    FirstName:string;
    LastName:string
    Age:number
}

// ShowEmployeeDetails expects an Employee object with FirstName, LastName and Age property
function ShowEmployeeDetails(emp:Employee){
    alert('hello ' + emp.FirstName + ' ' + emp.LastName + '. Your age is ' + emp.Age);
}

var emp1:Employee = {FirstName:'Bill',LastName:'Gates',Age:50};
var emp2 = 'is this an employee?'
var emp3 = {FirstName:'Bill',Age:50};

ShowEmployeeDetails(emp1); // Works as expected
ShowEmployeeDetails(emp2); // not an Employee object, shows undefined
ShowEmployeeDetails(emp3); // does not have LastName, shows undefined
```

Optional Property

Optional properties can be declared for an interface (using ?)
Optional properties need not be implemented

```
interface Employee{
    FirstName:string;
    LastName?:string // LastName is declared as optional property using ?
    Age:number
}

// ShowEmployeeDetails expects an Employee object with FirstName, LastName and Age property
function ShowEmployeeDetails(emp:Employee){
    alert('hello ' + emp.FirstName + ' ' + emp.LastName + '. Your age is ' + emp.Age);
}

var emp1:Employee = {FirstName:'Bill',LastName:'Gates',Age:50};
var emp2 = 'is this an employee?'
var emp3 = {FirstName:'Bill',Age:50};

ShowEmployeeDetails(emp1); // Works as expected
ShowEmployeeDetails(emp2); // not an Employee object, shows undefined
ShowEmployeeDetails(emp3); // Since LastName is now optional no error is shown
```

Interface


```
1  interface Employee{
2      name:string
3      basic:number
4      allowance:number
5  }
6
7  function getSalary(emp:Employee):number{
8      return emp.basic + emp.allowance
9  }
10
11 var emp1 = {name:"AC",basic:100, allowance:15}
12 var emp2 = {name:"AC",basic:100}
13 var emp3 = {name:"AC",basic:100, allowance:15,address:"India"}
14 var emp4 = {name:"AC",basic1:100, allowance2:15}
15
16 var emp5:Employee = {name:"AC",basic1:100, allowance2:15}
17
18 alert(getSalary(emp1))
19 alert(getSalary(emp2))
20 alert(getSalary(emp3))
21 alert(getSalary(emp4))
```

Mixin

Mixin

In object-oriented programming languages, a **mixin** is a class which contains a **combination of methods from other classes**.

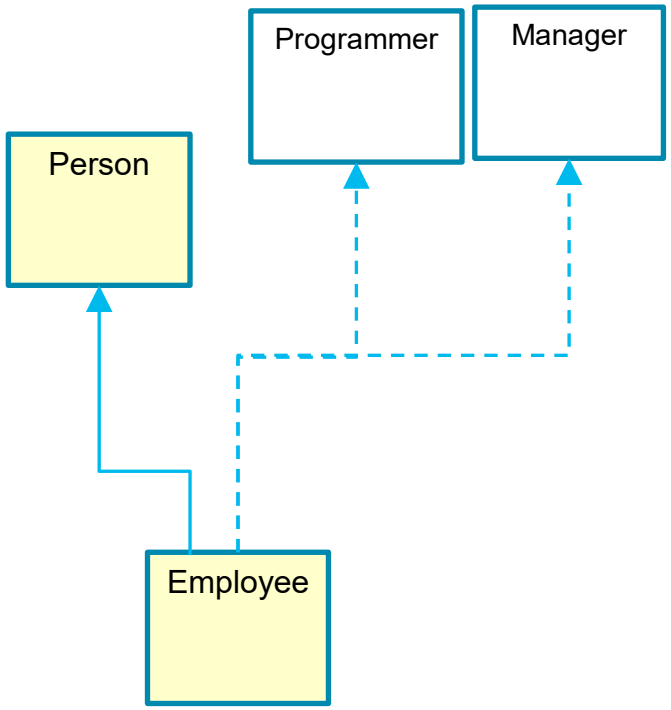
- Along with traditional OO hierarchies, another popular way of building up classes from reusable components is to build them by combining simpler partial classes – called **Mixin**
- Several languages support Mixin (e.g. Trait in PHP and Scala).
- This pattern in popular in JavaScript community, so TypeScript provides language support.

Mixins

```
// Base class (a class can inherit from only one base class)
class Person{
    constructor(public name:string, public age:number){
    }
    showInfo(){
        alert("Name: " + this.name + " Age: " + this.age);
    }
}

// First Mixin (a class can implement multiple mixin)
class Programmer{
    Code(){
        alert('Programmer.Code called');
    }
}

// First Mixin (a class can implement multiple mixin)
class Manager{
    Manage(){
        alert('Manager.Manage called');
    }
}
```



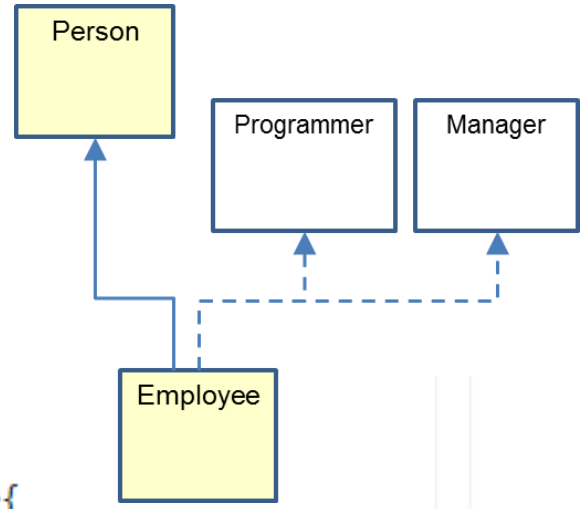
Mixins (Cont'd)

```
////////////////////////////////////////
// Library code needed to be added for Mixins
////////////////////////////////////////
function applyMixins(derivedCtor: any, baseCtors: any[]) {
    baseCtors.forEach(baseCtor => {
        Object.getOwnPropertyNames(baseCtor.prototype).forEach(name => {
            derivedCtor.prototype[name] = baseCtor.prototype[name];
        })
    });
}

// Child Class that inhertits/extends from Base class and
// implements multiple classes as mixins
class Employee extends Person implements Programmer, Manager{
    constructor(name, age, public salary:number){
        super(name,age); // super calls the constructor of base class
    }
    showInfo(){
        alert("Name:" + this.name + " Age:" + this.age + " Salary:" + this.salary);
    }

    Code: () => void
    Manage: () => void
}

applyMixins(Employee, [Programmer,Manager]);
```

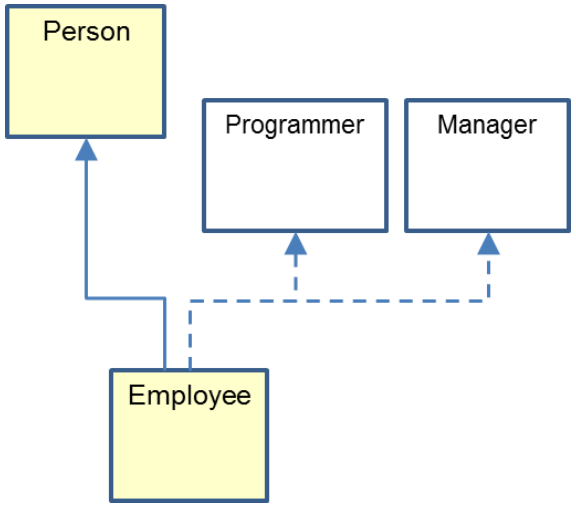


Mixins (Cont'd)

```
var per:Person = new Person('Aniruddha',40);
per.showInfo(); // calls showInfo of Person class

var emp = new Employee('Bill',55,100);
emp.showInfo(); // calls showInfo of Employee class

emp.Code();     // Calls Programmer.Code
emp.Manage();   // Calls Manager.Manage
```



JavaScript gochas (not fixed in TS)

- TypeScript does not introduce block scope (JavaScript only supports function scope)
- `;` is still optional in TypeScript also (`;` is not mandatory in JavaScript and it tries to infer `;` and sometime does it differently than expected)
- == vs === (and != vs !==)
  - == checks for value equality only
  - === checks for both type and value equality
- global variables (variables declared outside of function), implied global (variables declared within function without `var` keyword)
- issues with floating point (.1 + .2 != .3, it's something like .3000…00004)

Need for task automation

- Task automation is the use of technology to perform recurring tasks or processes where manual effort can be replaced.
- Here are the key reasons and benefits highlighting why task automation is essential:
- Increased Efficiency and Productivity
- Cost Reduction
- Reduced Human Error
- Scalability

Need for task automation

- Better Compliance and Auditing
- Employee Satisfaction
- Faster Decision-Making
- Availability and Uptime
- Competitive Advantage
- Integration Across Systems

Task Runners

- **Task Runners** are tools that automate repetitive tasks in software development workflows such as:
  - Minifying code
  - Compiling preprocessors (e.g., Sass, TypeScript)
  - Linting
  - Running tests
  - Bundling and optimizing assets
  - Watching files for changes
- They help streamline development and deployment processes, reduce human error, and save time.

Popular Task Runners

Task Runner	Language/Platform	Description
Grunt	JavaScript/Node.js	One of the earliest JavaScript task runners, uses plugins to run predefined tasks
Gulp	JavaScript/Node.js	Focuses on streaming and code-over-configuration, faster than Grunt
npm scripts	JavaScript/Node.js	Native way to define tasks directly in package.json, lightweight and widely used
Webpack	JavaScript	A module bundler that also handles tasks like transpilation, minification, and asset optimization
Make	C/C++, Unix-based	One of the oldest task automation tools, widely used in system-level programming

Popular Task Runners

Rake	Ruby	Build automation tool similar to Make, used in Ruby environments
Invoke / Fabric	Python	Pythonic way to manage task automation (Invoke for local, Fabric for remote tasks)
Gradle	Java/Kotlin	Modern build tool for JVM-based projects, supports task chaining and dependency management
MSBuild	.NET/C#	Microsoft's build system for .NET applications, integrated with Visual Studio

Common Use Cases

- Frontend Projects: Gulp or Webpack to compile Sass/Less, minify JS/CSS, reload browser
- Backend Projects: Gradle or MSBuild to compile, test, and deploy applications
- CI/CD Pipelines: Task runners invoked in Jenkins, GitLab CI, GitHub Actions
- DevOps Scripts: Use of Make, Ansible, or Fabric for server provisioning, deployments

Introduction to Gulp

- **Gulp** is a **JavaScript-based task runner** built on **Node.js** that automates development tasks like:
  - Minifying CSS/JS
  - Compiling Sass/LESS
  - Transpiling TypeScript
  - Live-reloading
  - Image optimization
  - Watching file changes
- It uses **code-over-configuration**, making it fast, flexible, and developer-friendly.

Why Gulp?

- Stream-based processing: Efficient file handling (no need to read/write intermediate files to disk)
- Fast execution using Node.js streams
- Simple syntax: Tasks written in plain JavaScript
- Large plugin ecosystem: Thousands of Gulp plugins for almost any task

Installing Gulp

- **1. Install Node.js and npm (if not already):**
- https://nodejs.org/
- 2. Initialize a Node project:
  - npm init –y
- 3. Install Gulp globally and locally:
  - npm install --global gulp-cli
  - npm install --save-dev gulp

Project Structure

Basic Gulpfile Example

- E:\Nodejs_ANZ\GulpDemoV1\gulpfile.js
- Running Gulp
  - Gulp
- This will:
  - Compile SCSS
  - Minify the CSS
  - Watch for changes and recompile automatically

Popular Gulp Plugins

Plugin	Use
gulp-sass	Compile SCSS to CSS
gulp-clean-css	Minify CSS
gulp-uglify	Minify JavaScript
gulp-imagemin	Compress images
browser-sync	Auto-reload browser on changes
gulp-babel	Transpile ES6+ to ES5

Benefits of Gulp

- Automates mundane tasks
- Faster builds via streaming
- Easy to integrate with any frontend stack
- Better file change handling with watch

Webpack

- **Webpack** is a **JavaScript module bundler** used primarily for **frontend development**.
- It **transforms, bundles, and optimizes** assets such as:
  - JavaScript files
  - CSS/SASS
  - Images
  - HTML templates

Webpack Key Concepts

Concept	Description
Entry	The starting point (main.js) for bundling.
Output	The bundled file location (usually dist/).
Loaders	Transformations on files (e.g., Babel for JS, css-loader).
Plugins	Extend Webpack (e.g., HtmlWebpackPlugin, MiniCssExtractPlugin).
Mode	development or production for optimizations.

Task Runner

- A **task runner** automates repetitive tasks like:
- Minifying CSS/JS
- Linting JavaScript
- Watching file changes
- Running tests
- Compiling SASS/LESS

Webpack vs Task Runners (Gulp/Grunt)

Feature	Webpack	Gulp / Grunt
Focus	Module bundling	General task automation
Output	Bundled modules	Output files, reports
Syntax	JavaScript config	JS + streams or plugins
Use Case	SPA, React/Vue/Angular	Asset pipeline, builds
Plugins/Loaders	Extensive ecosystem	Plugin-driven

Can Webpack Act as a Task Runner?

- Yes! Webpack can **replace** task runners for many use cases:
  - JS/CSS minification
  - Image optimization
  - File watching + hot reloading
  - Preprocessing with loaders
  - Production builds

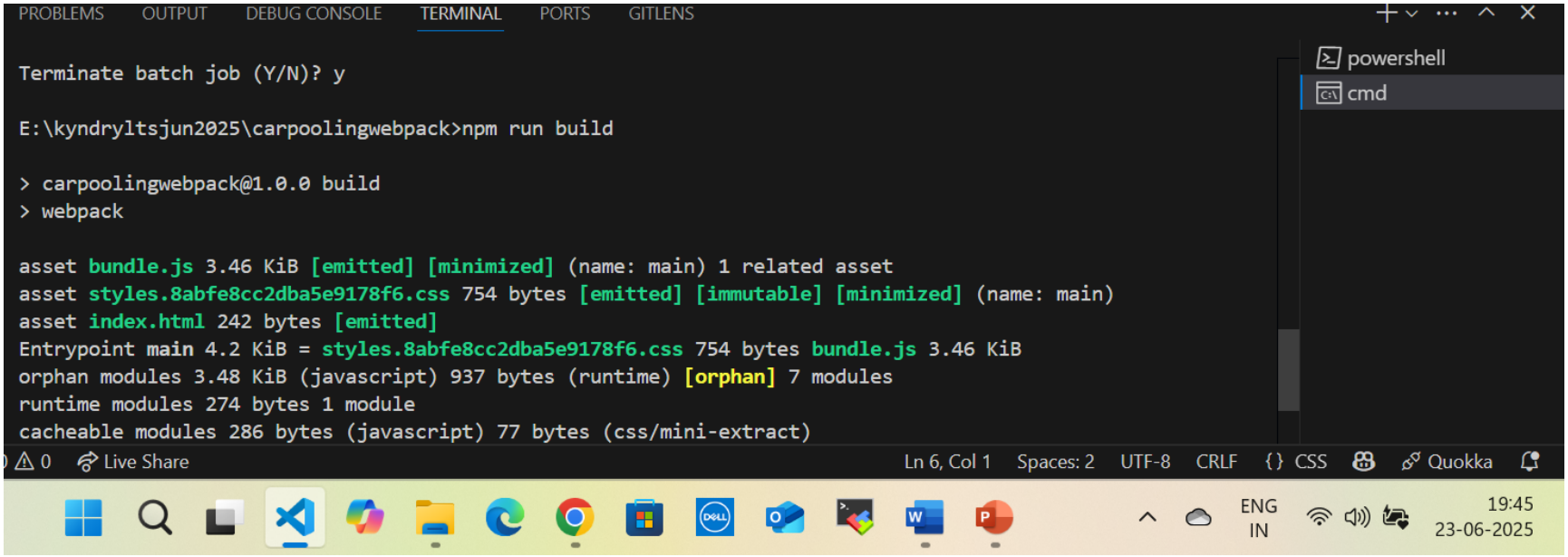
Webpack for typescript project


```
my-ts-webpack-app/
├── dist/
│   └── index.html
├── src/
│   └── index.ts
├── tsconfig.json
├── package.json
├── webpack.config.js
```

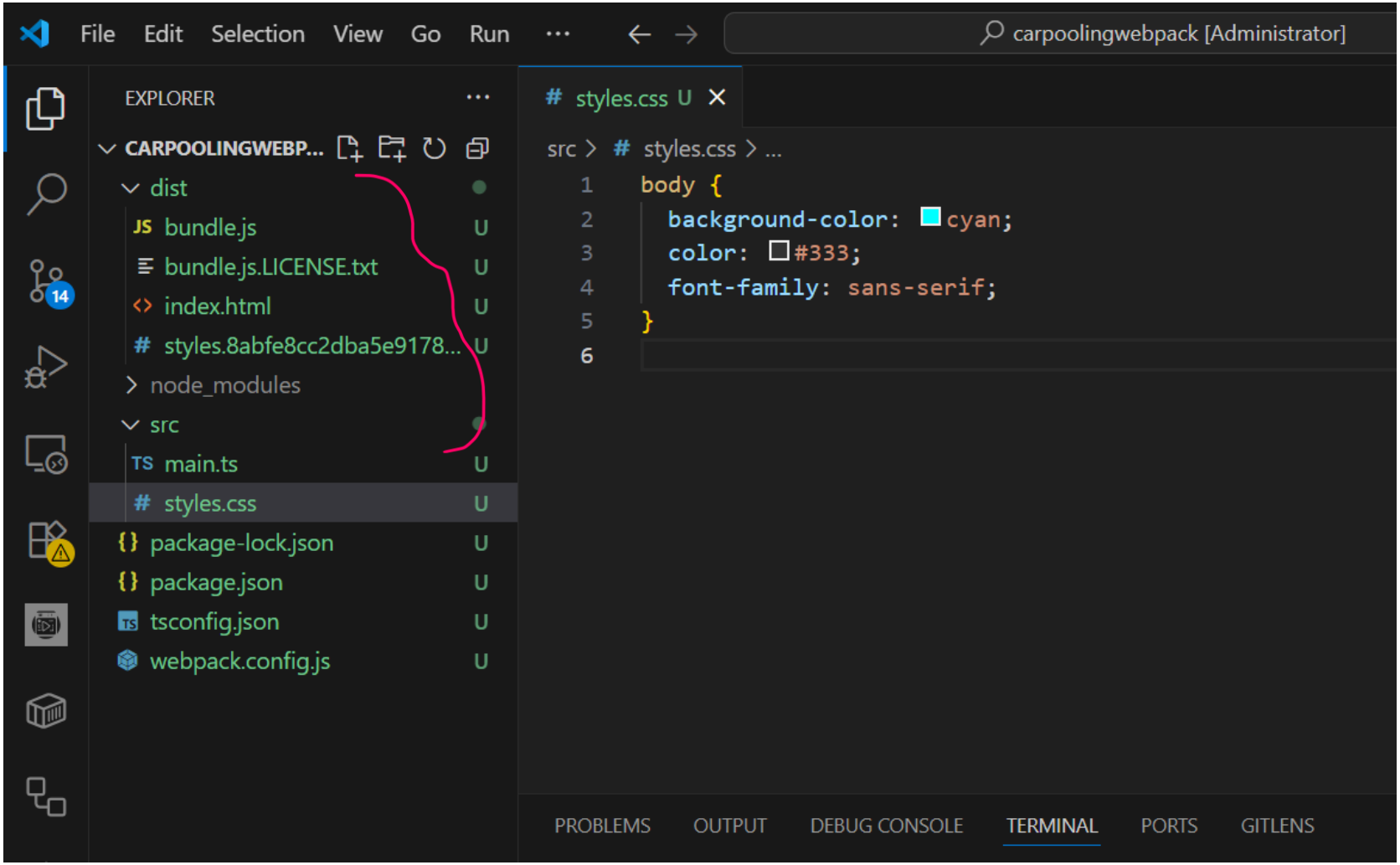
Install Required Packages

- npm install webpack webpack-cli webpack-dev-server --save-dev
- npm install typescript ts-loader --save-dev
- npm install html-webpack-plugin --save-dev

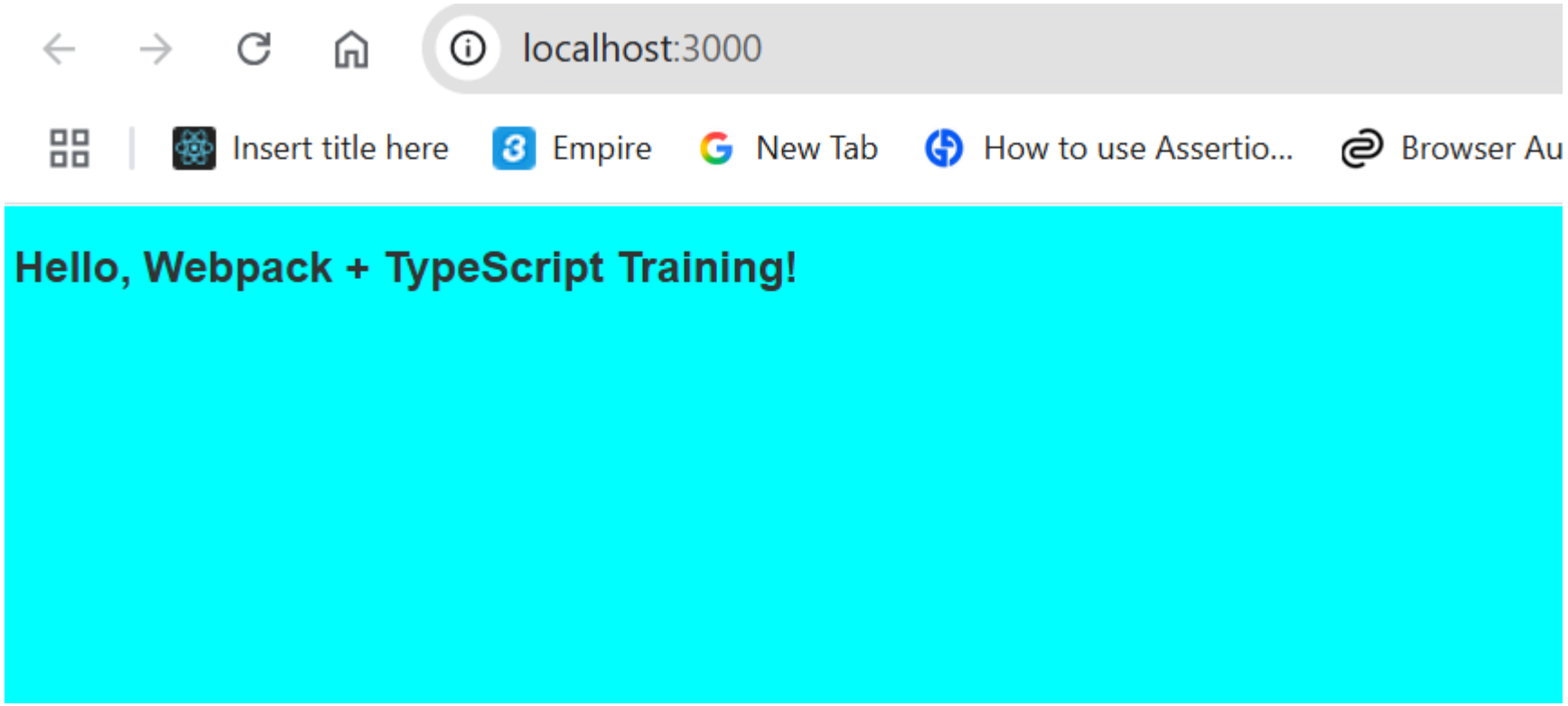
CSS Minification



CSS Minification



Npm start


localhost:3000
Insert title here
Empire
New Tab
How to use Assertio...
Browser Au
Hello, Webpack + TypeScript Training!

Result

- CSS is extracted and minified
- JavaScript is minified and optimized
- Files are hashed for cache busting
- dist/ is cleaned on each build

Image Optimization

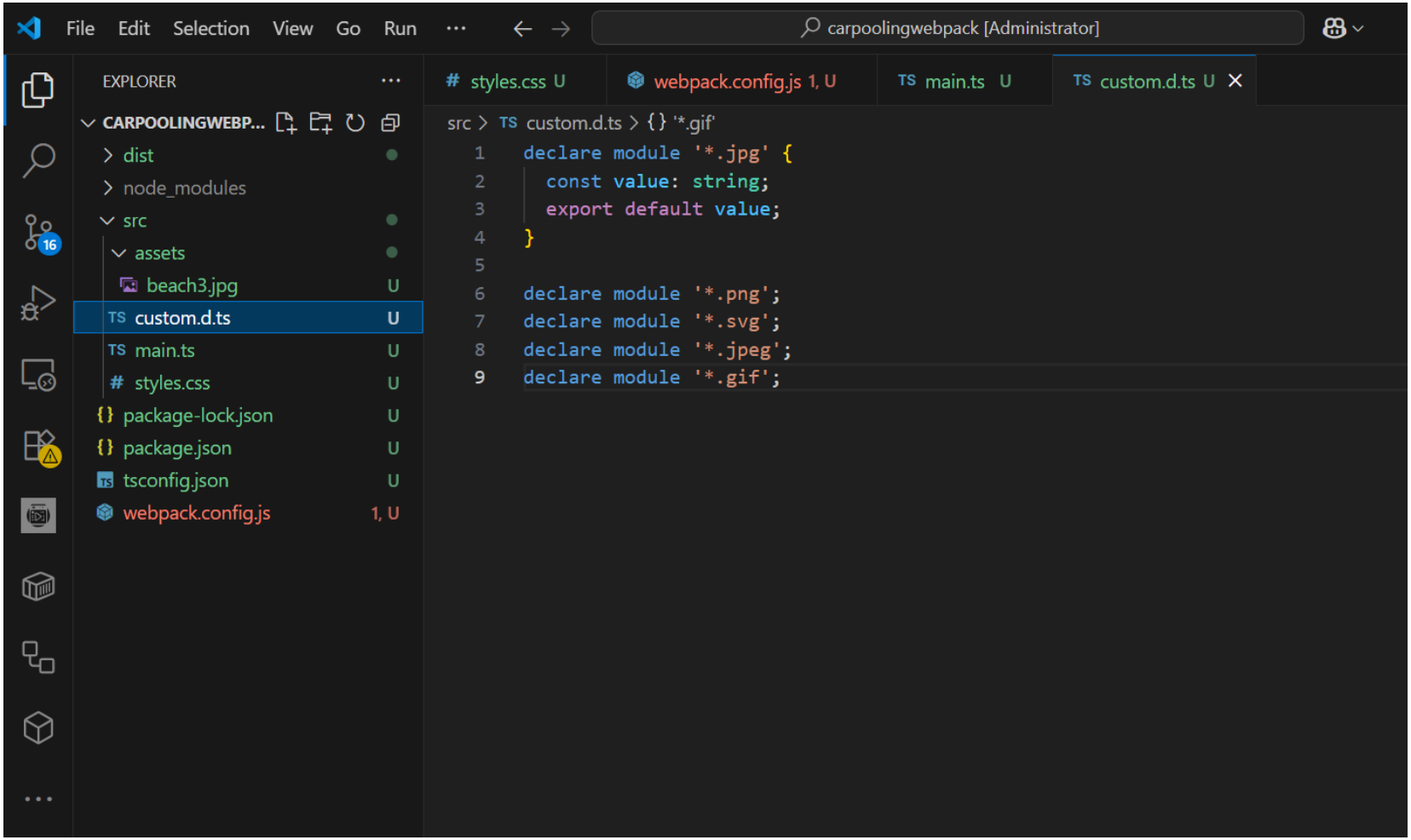


Image Optimization

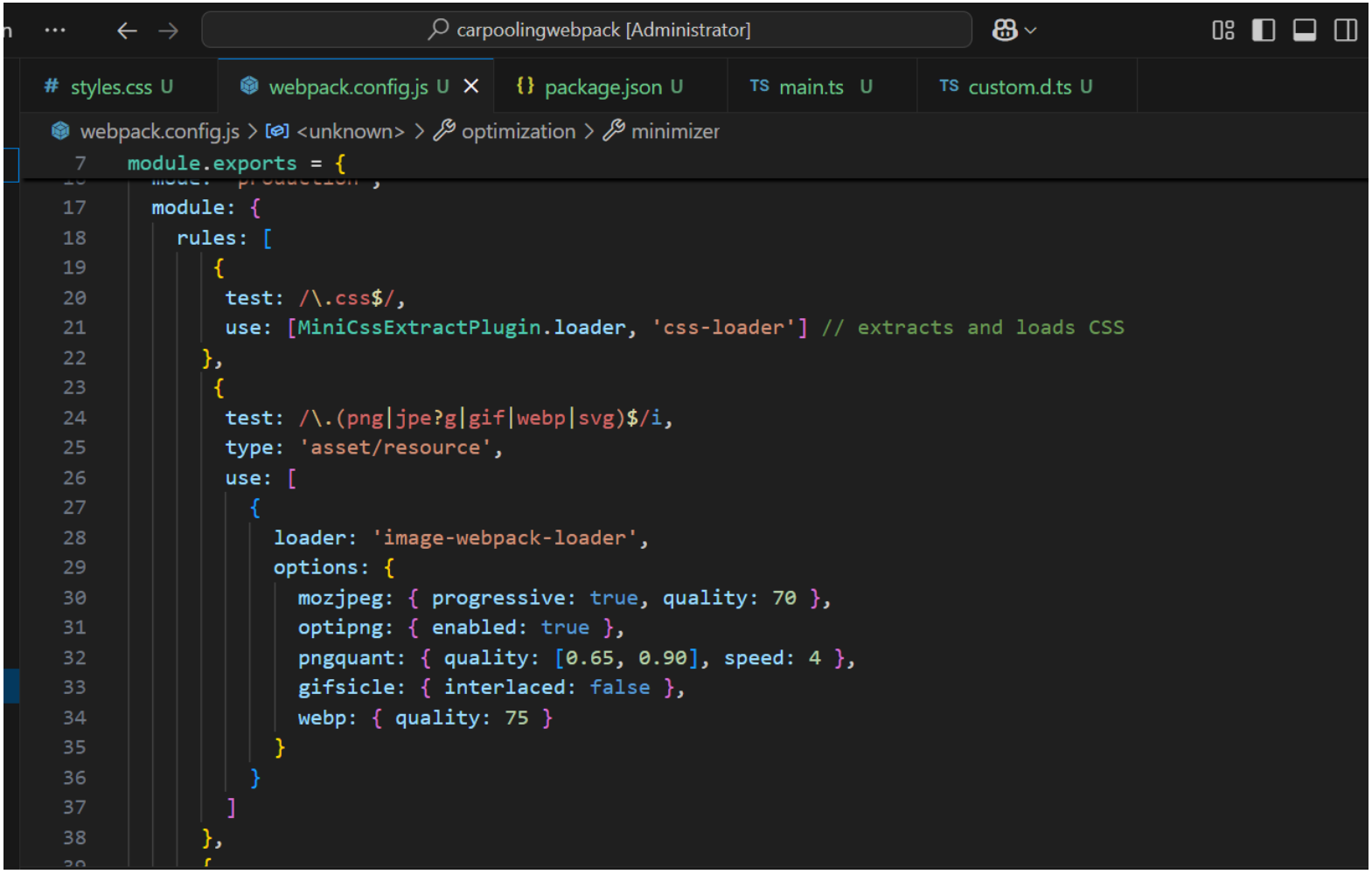


Image Optimization

Manual Testing vs. Automation Testing

Type	Description	Use When
Manual Testing	Tester executes tests without tools	UI validation, exploratory, short-term
Automation Testing	Tests run by scripts/tools like Selenium, JUnit, etc.	Regression, performance, large test sets

Black-Box vs. White-Box vs. Gray-Box

Approach	Visibility	Focus	Example Tools
Black-Box	No code access	Functional testing of outputs	Selenium, Postman
White-Box	Full code access	Internal logic, branches, paths	JUnit, Pytest, coverage
Gray-Box	Partial code access	Combine UI + logic understanding	Cypress, API+DB test

Levels of Testing

Level	Purpose	Who Performs
Unit Testing	Validate smallest parts (functions/classes)	Developers (TDD, CI)
Integration	Test interaction between components	Developers/Testers
System Testing	Test whole system end-to-end	QA/Test Team
Acceptance Testing	Business-level validation (UAT)	Product Owners/Clients

Testing Types Based on Purpose

Type	Focus	Common Tools
Functional	Features behave as expected	Selenium, Cypress, Postman
Non-Functional	Performance, security, usability	JMeter, OWASP ZAP, Lighthouse
Regression	No existing feature is broken	Selenium, TestNG, Playwright
Smoke/Sanity	Quick check of core functions	Any test suite
End-to-End (E2E)	Full user flow validation	Cypress, Playwright
Exploratory	Ad hoc, human-intuition-based	Manual

Agile/DevOps-Oriented Testing Approaches

Approach	Description
Test-Driven Development (TDD)	Write tests before writing code
Behavior-Driven Development (BDD)	Use natural language (e.g. Gherkin/Cucumber) to define tests
Continuous Testing	Automated tests run as part of CI/CD pipelines
Shift-Left Testing	Testing starts early in the development process
Shift-Right Testing	Focus on testing in production (monitoring, chaos tests)

Testing Frameworks – JS/TS

Framework	Type	Notes
Jest	Unit, Snapshot	Great for React/Vue/TS projects, fast CI
Mocha + Chai	Unit, Integration	Flexible, modular
Cypress	E2E, UI	All-in-one for browser automation
Playwright	E2E, Cross-browser	Fast, modern alternative to Cypress
Vitest	Unit	Vite-native Jest alternative

Testing Frameworks - Python

Framework	Type	Notes
unittest	Unit	Python built-in testing module
pytest	Unit, Integration	Most popular, simple & powerful
robotframework	Acceptance, UI	Keyword-driven, readable for QA
behave	BDD	Gherkin-style behavior tests
Locust	Load testing	Simple performance/load testing tool

Testing Frameworks - Java

Framework	Type	Notes
JUnit 5	Unit, Integration	Modern, annotation-driven testing
TestNG	Unit, Data-driven	Flexible configuration, parallel tests
Cucumber-JVM	BDD	Gherkin-based test specs
RestAssured	API Testing	Fluent API for REST testing
Selenium WebDriver	UI, E2E	Browser automation

Testing Frameworks - .NET

Framework	Type	Notes
xUnit	Unit	Modern, .NET Core compatible
NUnit	Unit	Older, still widely used
SpecFlow	BDD	Cucumber for .NET
MSTest	Unit	Microsoft official framework
Playwright.NET	UI, E2E	Cross-browser UI automation

Testing Frameworks – PHP and Ruby

Framework	Type	Notes
PHPUnit	Unit	De facto testing standard for PHP
Codeception	E2E, BDD	Combines unit, API, acceptance testing

Framework	Type	Notes
RSpec	Unit, BDD	Very expressive syntax
Cucumber-Ruby	BDD	Gherkin-based
Capybara	UI, E2E	Browser automation with RSpec

Testing Frameworks

Framework	Type	Notes
PHPUnit	Unit	De facto testing standard for PHP
Codeception	E2E, BDD	Combines unit, API, acceptance testing

Framework	Type	Notes
RSpec	Unit, BDD	Very expressive syntax
Cucumber-Ruby	BDD	Gherkin-based
Capybara	UI, E2E	Browser automation with RSpec

Choosing the Right Testing Framework

Project Type	Recommended Framework(s)
React/Angular UI	Jest, Cypress, Playwright
Backend API	SuperTest, REST-assured, Postman, pytest
Microservices	TestNG, Pytest, xUnit
CI/CD pipelines	Jest, Mocha, Postman (via Newman)
BDD/UAT	Cucumber, SpecFlow, Behave
Load Testing	JMeter, Locust, Artillery

What is BDD?

- BDD (Behavior-Driven Development) encourages:
  - Writing tests in terms of "behavior" instead of implementation
  - Using human-readable language like describe, it, should, when, then
  - Aligning tests with user stories or expected outcomes

Type Compatibility

- Interface Compatibility
- interface Person {
- name: string;
- age: number;
- }

- const user = { name: 'Alice', age: 30, city: 'Delhi' };

- const person: Person = user; // ✅ OK – extra property is fine

Type Compatibility

- Interface Compatibility
- interface Person {
- name: string;
- age: number;
- }

- const user = { name: 'Alice', age: 30, city: 'Delhi' };

- const person: Person = user; // ✅ OK – extra property is fine

Type Compatibility

- Function Compatibility
- Parameters: Less is okay
- type Func1 = (a: number, b: number) => void;
- type Func2 = (a: number) => void;

- const f1: Func1 = (a) => {}; // ✅ OK: Extra param is ignored
- const f2: Func2 = (a, b) => {}; // ❌ Error: Too many required args

Type Compatibility

- Function Compatibility
- Return Type: More specific is fine
- type A = () => string;
- type B = () => string | number;

- const a: A = () => "hello";
- const b: B = a; // ✅ OK – string is assignable to string | number

Type Compatibility

- Class Compatibility
- class Car {
- wheels = 4;
- }

- class Truck {
- wheels = 4;
- load = 100;
- }

- const c: Car = new Truck(); // ✅ OK

Type Compatibility

- Enum Compatibility
- enum Direction { Up, Down }

- let d: Direction = Direction.Up;
- d = 1; // ✅ Enums are compatible with numbers
- d = Direction.Down;

Type Compatibility

- enum Status { Ready, Waiting }

- let s: Status;
- // s = Direction.Up; // ❌ Not compatible with different enums

Type Compatibility

- Generic Compatibility
- interface Box<T> {
- value: T;
- }

- let box1: Box<number> = { value: 123 };
- let box2: Box<string> = { value: 'abc' };

- // box1 = box2; // ❌ Not compatible – T is different

Type Compatibility

- Incompatible Types
- interface Box<T> {
- value: T;
- }

- let box1: Box<number> = { value: 123 };
- let box2: Box<string> = { value: 'abc' };

- // box1 = box2; // ❌ Not compatible – T is different

Type Inference

- Type inference is when TypeScript automatically figures out the type of a variable, function return, or expression without explicit annotation.
- Variable Type Inference
- let name = "Alice"; // inferred as `string`
- let age = 30; // inferred as `number`
- let isAdmin = true; // inferred as `boolean`

Type Inference

- Function Return Type Inference
- function greet(user: string) {
- return `Hello, ${user}`;
- }
- Inferred return type: string

Type Inference

- Array and Object Inference
- let scores = [100, 98, 85]; // inferred as number[]
- let user = {
- name: "Alice",
- age: 30,
- }; // inferred as { name: string; age: number }

Type Inference

- Destructuring Inference
- const person = { name: "Bob", age: 25 };
- const { name, age } = person; // inferred: name:string, age:number

Type Inference

- Contextual Typing
- Type inferred from surrounding context:
  - window.addEventListener("click", (e) => {
    console.log(e.clientX); // `e` is inferred as MouseEvent});

Union Types

- Union types let you define a variable that can hold more than one type.
- It's one of TypeScript’s core features for flexibility and type safety.
- let value: string | number;
- Now value can be either a string or a number.

Union Types

- 1. Assigning Different Types
- value = 'Hello'; // ✅ string
- value = 123; // ✅ number
- value = true; // ❌ Error: boolean not allowed

Union Types

- 2. Function with Union Parameter
- function printId(id: string | number) {
- console.log('Your ID is: ' + id);
- }

- printId(101); // ✅
- printId("AB-321"); // ✅

Union Types

- Type Narrowing
- To safely use type-specific methods, narrow the union:
- function format(input: string | number) {
- if (typeof input === "string") {
- return input.toUpperCase(); // ✅ OK for string
- }
- else {
- return input.toFixed(2); // ✅ OK for number
- }}

Union Types

- Union with Custom Types

```
type Admin = { role: "admin"; accessLevel: number };
type User = { role: "user"; email: string };

type Person = Admin | User;

function describe(person: Person) {
  if (person.role === "admin") {
    console.log("Access Level:", person.accessLevel);
  } else {
    console.log("Email:", person.email);
 }
}
```

Union Types

Union in Arrays

```
let data: (string | number)[];
data = ['a', 1, 'b', 2]; // ✅ OK
```

Union Types

Union Return Types

```
function getScore(score: number): string | number {
  return score > 40 ? "Pass" : score; // can return string or number
}
```

Promises

- A **Promise** is an object that represents the **eventual result** (or failure) of an asynchronous operation.

```
const promise = new Promise((resolve, reject) => {
  // async task
  if (success) {
    resolve(result);
  } else {
    reject(error);
  }
});
```

Promise States

State	Meaning
`pending`	Initial state, not fulfilled yet
`fulfilled`	Operation completed successfully
`rejected`	Operation failed

Consuming Promises


```
fetch('https://api.example.com')
  .then(response => response.json())  // on success
  .then(data => console.log(data))
  .catch(err => console.error('Error:', err))  // on failure
  .finally(() => console.log('Done!')); // always runs
```

Creating Your Own Promise


```ts
function getData(): Promise<string> {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve("Hello from Promise");
    }, 1000);
  });
}
```

async/await with Promises


```ts
async function main() {
  try {
    const message = await getData();
    console.log(message);
  } catch (err) {
    console.error(err);
  }
}
```

> `await` unwraps the promise, `try/catch` handles errors.

Chaining Promises

```
doStep1()
  .then(result1 => doStep2(result1))
  .then(result2 => doStep3(result2))
  .catch(error => console.error(error));
```

Promise.all – Run in Parallel


```
Promise.all([
  fetch('/api/user'),
  fetch('/api/posts')
])
.then(([userRes, postRes]) => {
  // Both results available here
});
```

Promise.race – First one wins

```
Promise.race([
  fetch('/slow'),
  fetch('/fast')
])
.then(result => console.log('First response:', result));
```

Promises in TypeScript


```typescript
function delay(ms: number): Promise<string> {
  return new Promise(resolve => {
    setTimeout(() => resolve("Done"), ms);
  });
}
```

Summary

Feature	Purpose
new Promise()	Create a promise
.then()	Handle success
.catch()	Handle errors
.finally()	Always runs (cleanup, logging, etc.)
async/await	Write promise-based code synchronously
Promise.all()	Wait for multiple promises

Importance of Async

- async functions are a modern, elegant way to write asynchronous code — making it easier to read, write, debug, and maintain, especially in real-world apps involving:
  - APIs & HTTP calls
  - File operations
  - Timers,
  - Delays
  - User interaction
  - Background tasks

Why async Matters

- Simplifies Promises

```
fetch(url)
  .then(res => res.json())
  .then(data => console.log(data))
  .catch(err => console.error(err));
```

Why async Matters

- Simplifies Promises by Use async/await:

```
async function getData() {
  try {
    const res = await fetch(url);
    const data = await res.json();
    console.log(data);
  } catch (err) {
    console.error(err);
  }
}
```

Why async Matters

- Improves Readability & Flow
- Asynchronous code reads like synchronous code, making it:
  - Easier to follow
  - Less nested
  - More maintainable

```
async function loadPage() {
  await connectToServer();
  await fetchUserData();
  await renderDashboard();
}
```

Why async Matters

- Better Error Handling with try/catch

```
try {
  const data = await fetchSomething();
} catch (err) {
  console.error("Something went wrong:", err);
}
```

Ideal for Real-World Use Cases

Use Case	Why async/await Helps
API Calls	Cleanly await fetch/post/put/delete
File Uploads/Downloads	Pause until response completes
Database Queries	Async DB drivers (Mongo, Prisma, etc.)
WebSocket / Realtime events	Await reconnection, retries
Browser Interactions	Wait for UI actions before proceeding

Generators

- Generators are special functions that can pause and resume execution, making them ideal for:
  - Iteration over custom sequences
  - Lazy evaluation
  - Asynchronous control flow
  - Complex state machines

Generators


```
function* countToThree() {
  yield 1;
  yield 2;
  yield 3;
}

const counter = countToThree();

console.log(counter.next()); // { value: 1, done: false }
console.log(counter.next()); // { value: 2, done: false }
console.log(counter.next()); // { value: 3, done: false }
console.log(counter.next()); // { value: undefined, done: true
}
```

Looping a Generator


```
for (const num of countToThree()) {
  console.log(num); // 1, 2, 3
}
```

Infinite Generator


```
function* infiniteCounter() {
  let i = 0;
  while (true) {
    yield i++;
  }
}

const gen = infiniteCounter();
console.log(gen.next().value); // 0
console.log(gen.next().value); // 1
console.log(gen.next().value); // 2
```

Passing Values into Generator


```ts
function* greet() {
  const name = yield "What is your name?";
  yield `Hello, ${name}!`;
}

const g = greet();
console.log(g.next().value);        // "What is your name?"
console.log(g.next("Alice").value); // "Hello, Alice!"
```

Using Generators with yield* (Delegation)


```ts
function* inner() {
  yield "a";
  yield "b";
}

function* outer() {
  yield* inner(); // delegate to inner()
  yield "c";
}

for (const val of outer()) {
  console.log(val); // a, b, c
}
```

Generator with TypeScript


```ts
function* generateIds(): Generator<number> {
  let id = 1;
  while (true) {
    yield id++;
  }
}
```

Async Generators (async function*)

```
async function* fetchPages() {
  let page = 1;
  while (page <= 3) {
    const res = await fetch(`/api/data?page=${page}`);
    yield await res.json();
    page++;
  }
}
```

Usage:

ts

```
for await (const data of fetchPages()) {
  console.log(data);
}
```

Summary

Feature	Benefit
`yield`	Emits a value, then pauses
`next()`	Resumes execution
`yield*`	Delegates to another generator
`async function*`	Emits promises in async iteration
TypeScript `Generator<T>`	Adds strong typing

Design Pattern

- 1) Creational Pattern
  - Factory Method Pattern
  - Abstract Factory Pattern
  - Singleton Pattern
  - Prototype Pattern
  - Builder Pattern
  - Object Pool Pattern

Design Pattern

- 2) Structural Pattern
  - Adapter Pattern
  - Bridge Pattern
  - Composite Pattern
  - Decorator Pattern
  - Facade Pattern
  - Flyweight Pattern
  - proxy Pattern

Design Pattern

- 3) Behavioral Pattern
  - Chain of Responsibility
  - Command Pattern
  - Interpreter Pattern
  - Iterator Pattern
  - Mediator Pattern
  - Memento Pattern
  - Observer Pattern
  - State Pattern
  - Strategy Pattern
  - Template Pattern

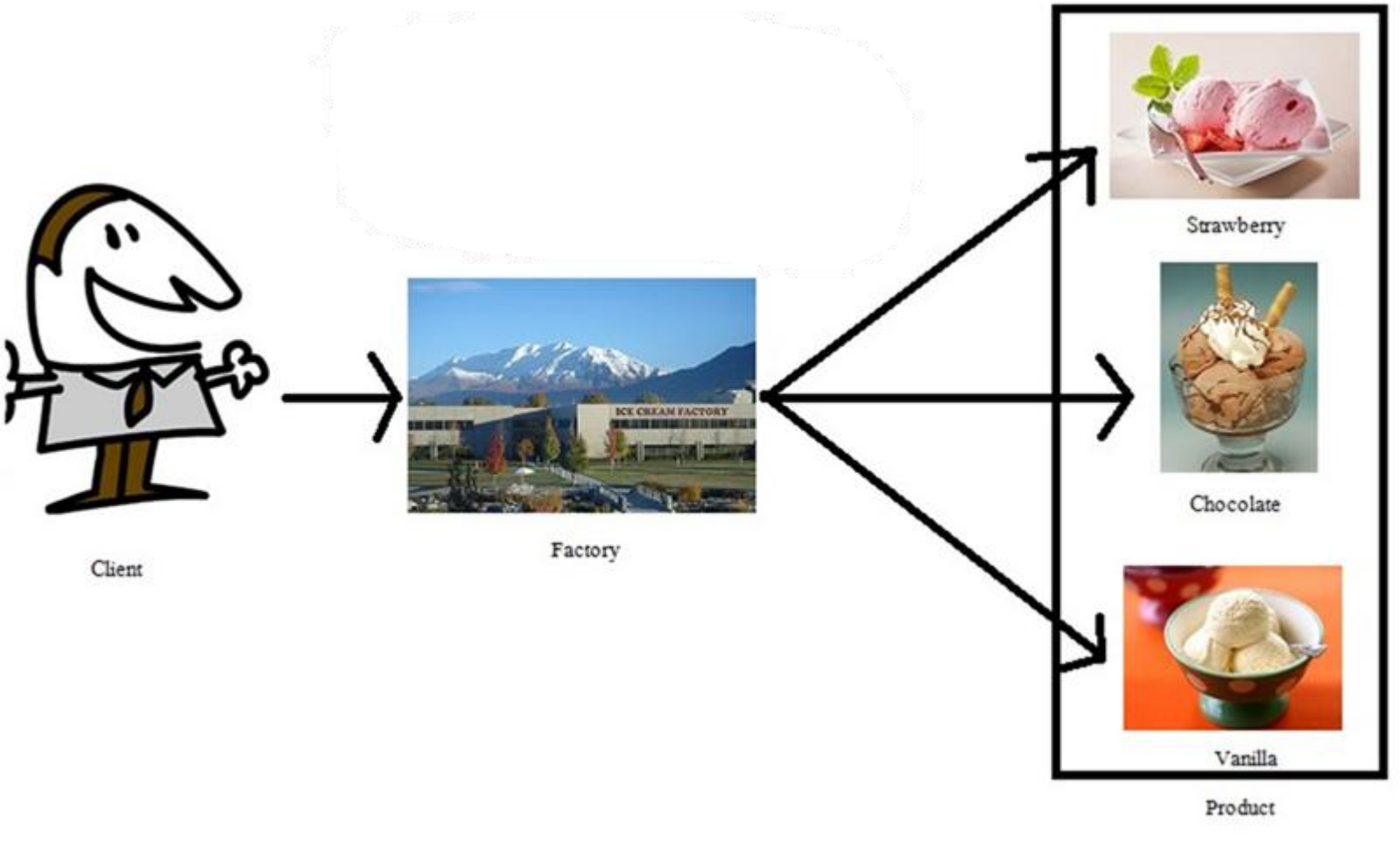
Typescript creational patterns

Constructor Pattern

```
class User {
  constructor(
    public username: string,
    public email: string,
    public role: string = "viewer"
  ) {}
}

// Used after a form is submitted:
const newUser = new User("alice", "alice@example.com");
console.log(newUser); // { username: 'alice', email: 'alice@example.com', role: 'viewer' }
```

Factory Design Pattern



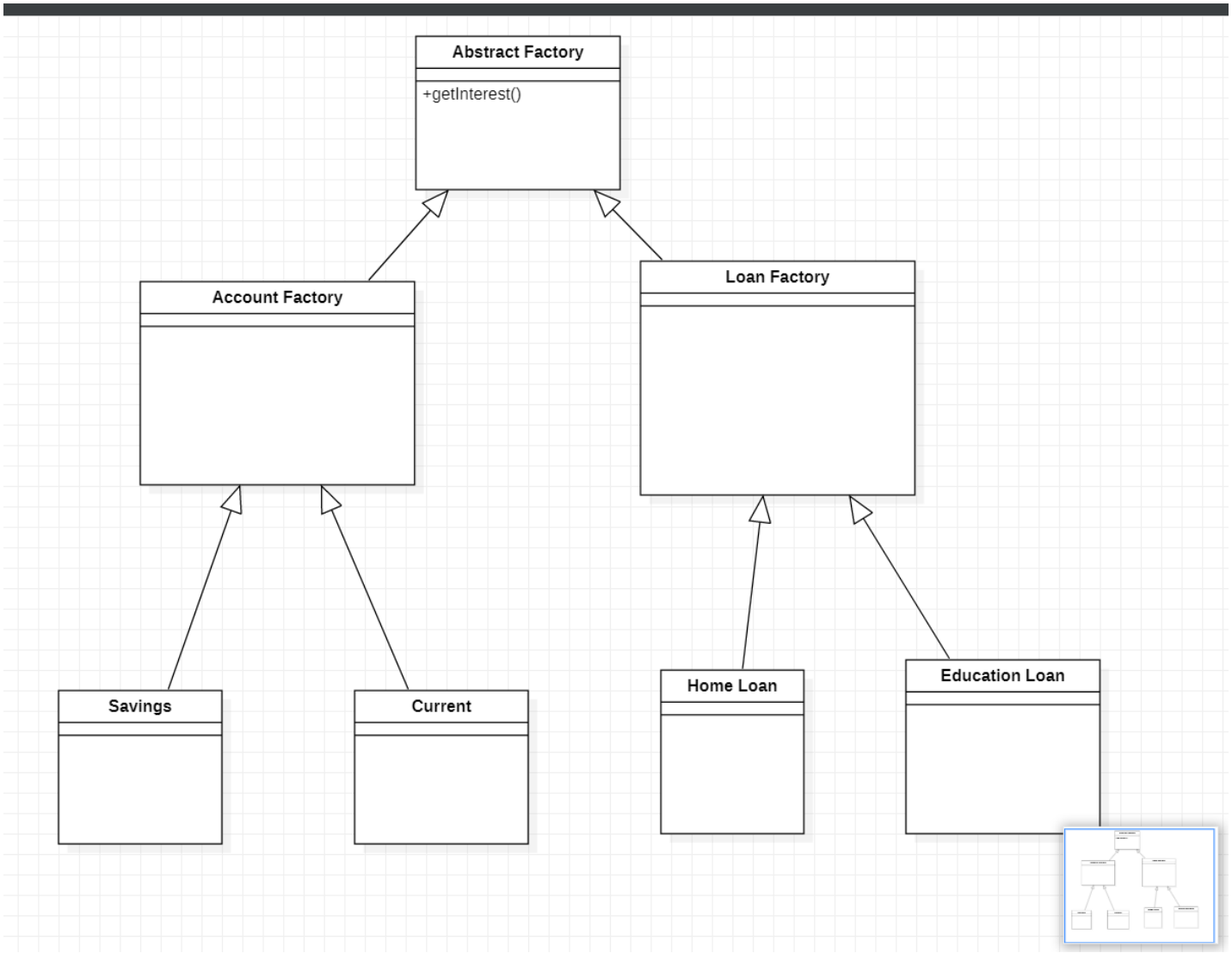
Typescript creational patterns

Factory Pattern

```
class APIClientFactory {
  static getClient(type: "rest" | "graphql"): APIClient {
    if (type === "rest") return new RESTClient();
    return new GraphQLClient();
  }
}

const client = APIClientFactory.getClient("graphql");
```

The Abstract Factory pattern



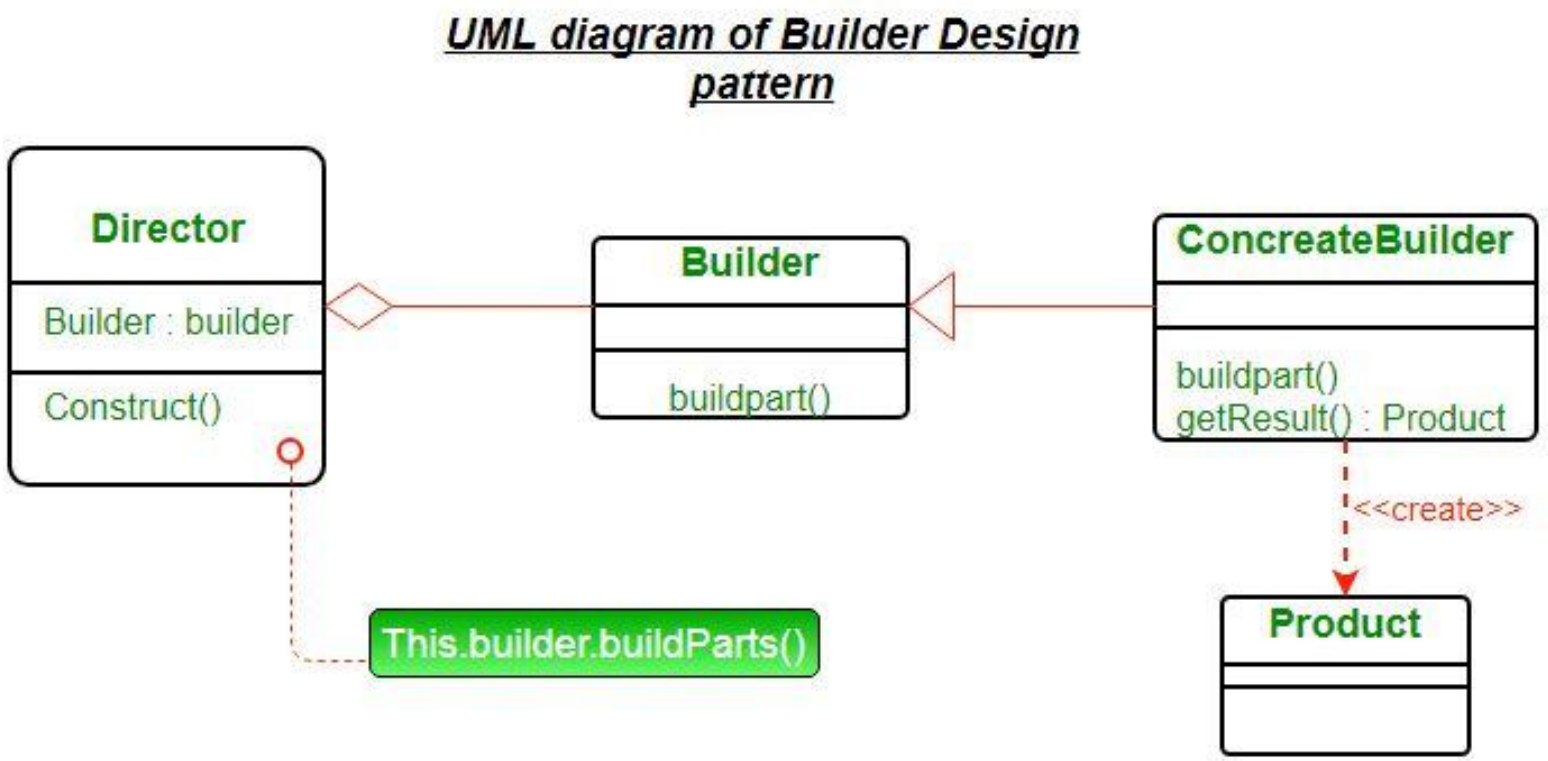
Abstract Factory Pattern

```
interface ThemeFactory {
  createButton(): ThemeButton;
}

class DarkThemeFactory implements ThemeFactory {
  createButton(): ThemeButton {
    return new DarkButton();
  }
}

class LightThemeFactory implements ThemeFactory {
  createButton(): ThemeButton {
    return new LightButton();
  }
}
```

Builder Design Pattern



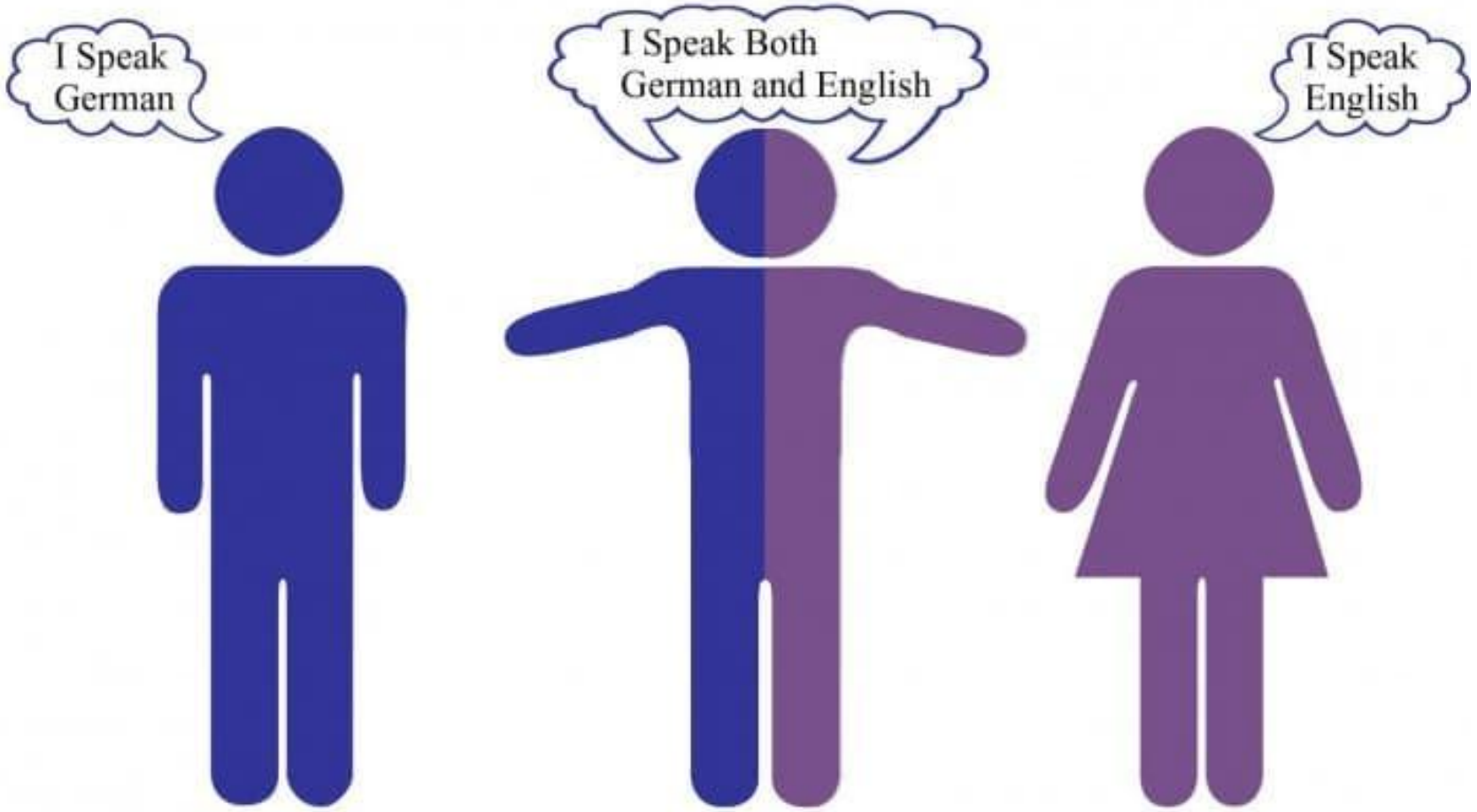
Typescript creational patterns

Builder Pattern

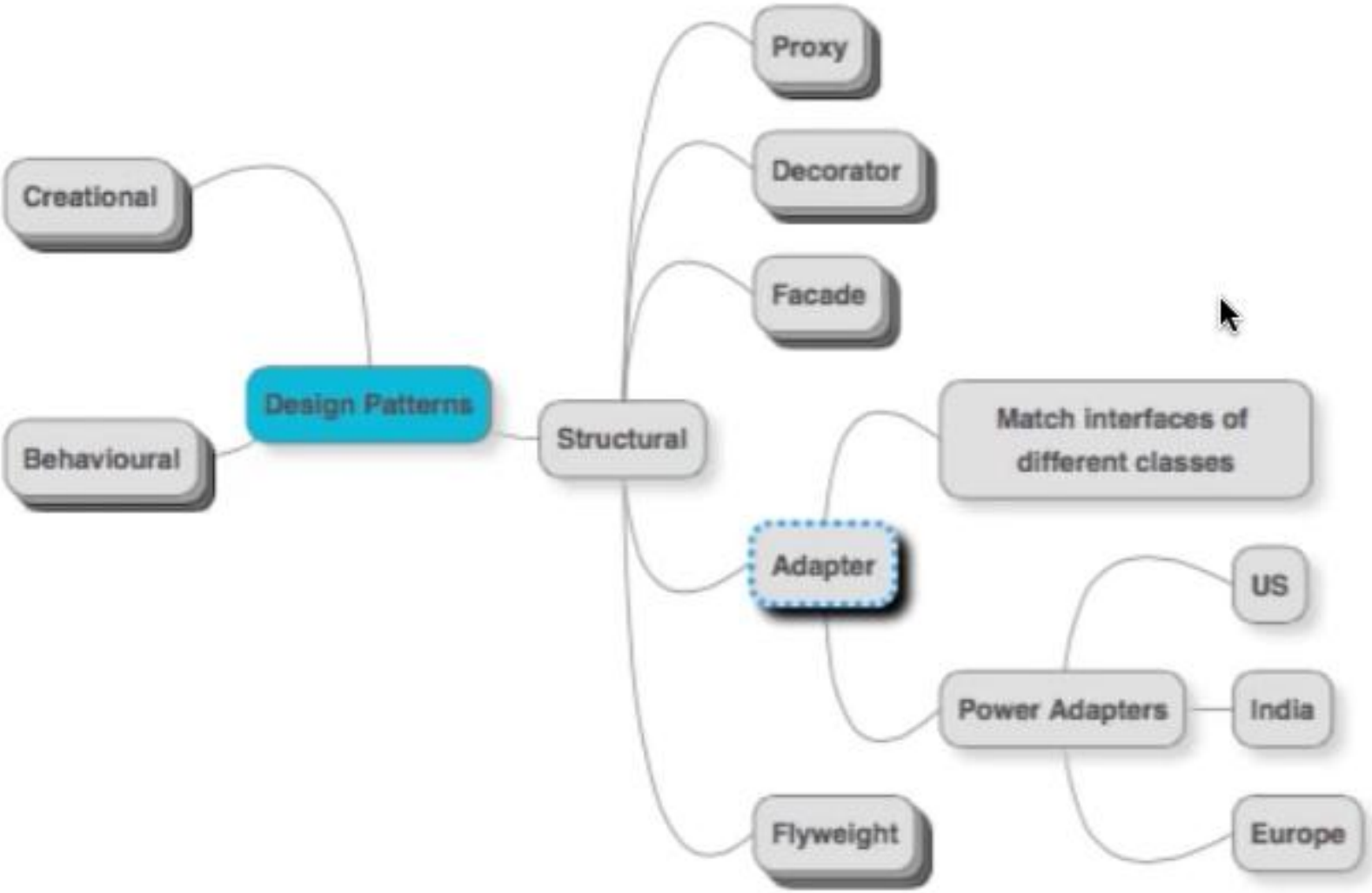
```
const query = new SearchQueryBuilder()
  .setQuery("laptops")
  .addFilter("brand:Apple")
  .addFilter("price:<2000")
  .setSortBy("price")
  .build();

console.log(query.toString());
// Output: Query: laptops, Filters: [brand:Apple, price:<2000], Sort: price
```

Adapter Pattern



Adapter Pattern



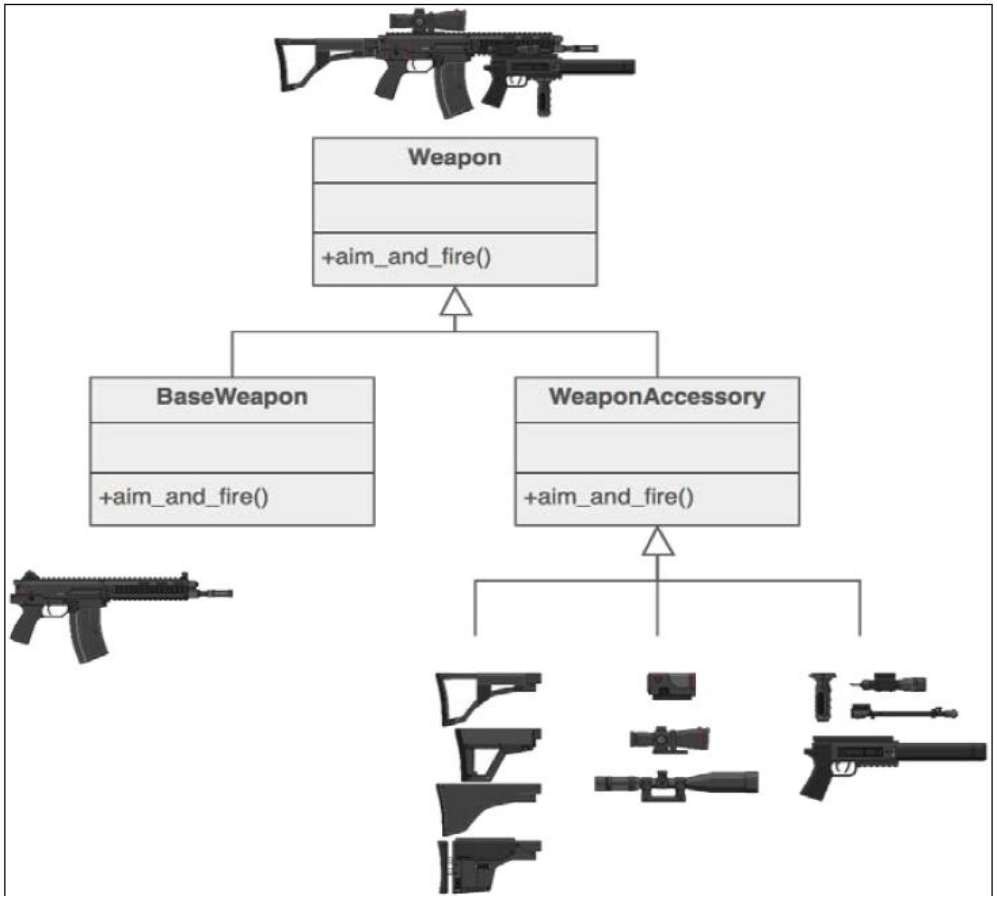
The Decorator Pattern

- A **Decorator** pattern can add responsibilities to an object dynamically, and in a transparent manner.
- This defines additional responsibilities for an object at runtime or dynamically. We add certain attributes to objects with an interface.
- Real examples of such extensions are: adding a silencer to a gun, using different camera lenses (in cameras with removable lenses), and so on.
- The Decorator pattern shines when used for implementing **cross-cutting concerns**

The Decorator Pattern

- Data validation
- Transaction processing (A transaction in this case is similar to a database transaction, in the sense that either all steps should be completed successfully, or the transaction should fail.)
- Caching
- Logging
- Monitoring
- Debugging
- Business rules
- Compression
- Encryption

The Decorator Pattern



Façade Design Pattern

- The façade is generally referred to as the face of the building, especially an attractive one.
- It can be also referred to as a behavior or appearance that gives a false idea of someone's true feelings or situation.
- When people walk past a façade, they can appreciate the exterior face but aren't aware of the complexities of the structure within. This is how a façade pattern is used.
- Façade hides the complexities of the internal system and provides an interface to the client that can access the system in a very simplified way.

Façade Design Pattern

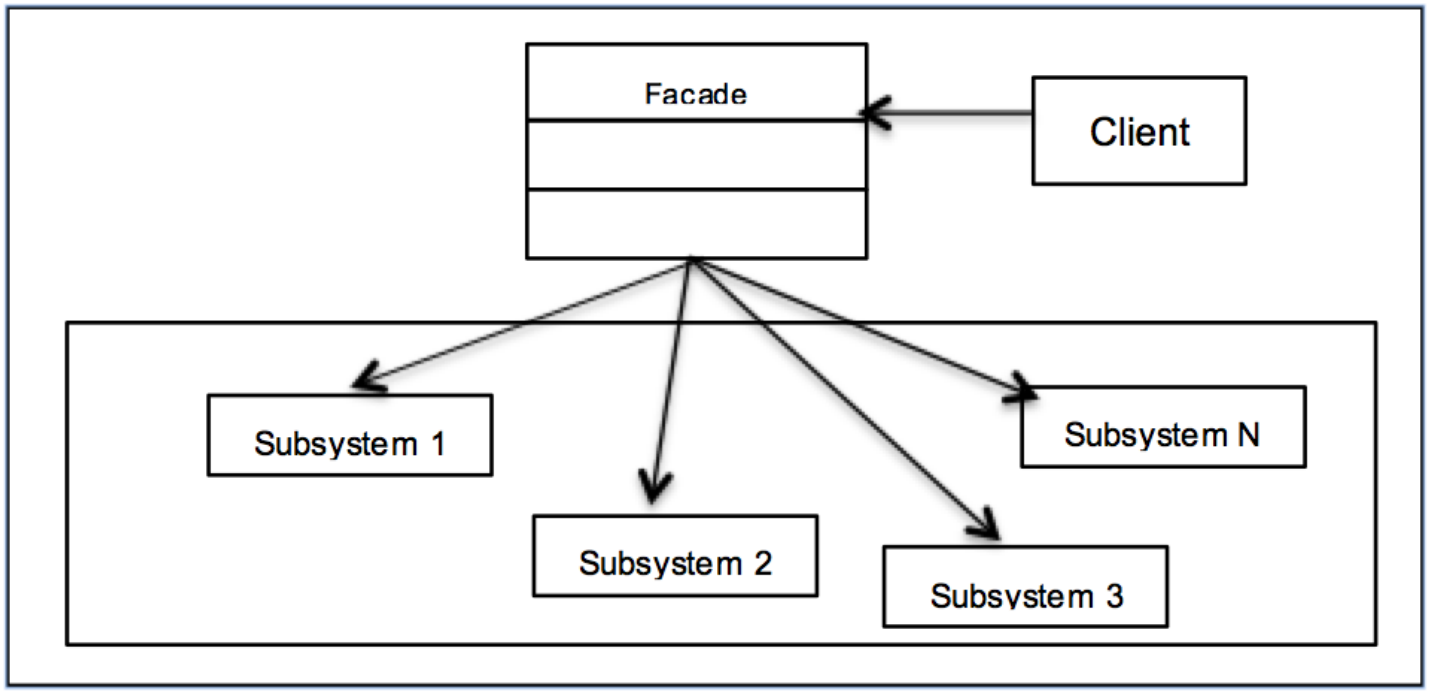
- Consider the example of a storekeeper. Now, when you, as a customer, visit a store to buy certain items, you're not aware of the layout of the store.
- You typically approach the storekeeper, who is well aware of the store system. Based on your requirements, the storekeeper picks up items and hands them over to you.
- Isn't this easy? The customer need not know how the store looks and s/he gets the stuff done through a simple interface, the storekeeper.

Façade Design Pattern

The Façade design pattern essentially does the following:

- It provides a unified interface to a set of interfaces in a subsystem and defines a high-level interface that helps the client use the subsystem in an easy way.
- Façade discusses representing a complex subsystem with a single interface object.
- It doesn't encapsulate the subsystem but combines the underlying subsystems.
- It promotes the decoupling of the implementation with multiple clients.

Façade Design Pattern



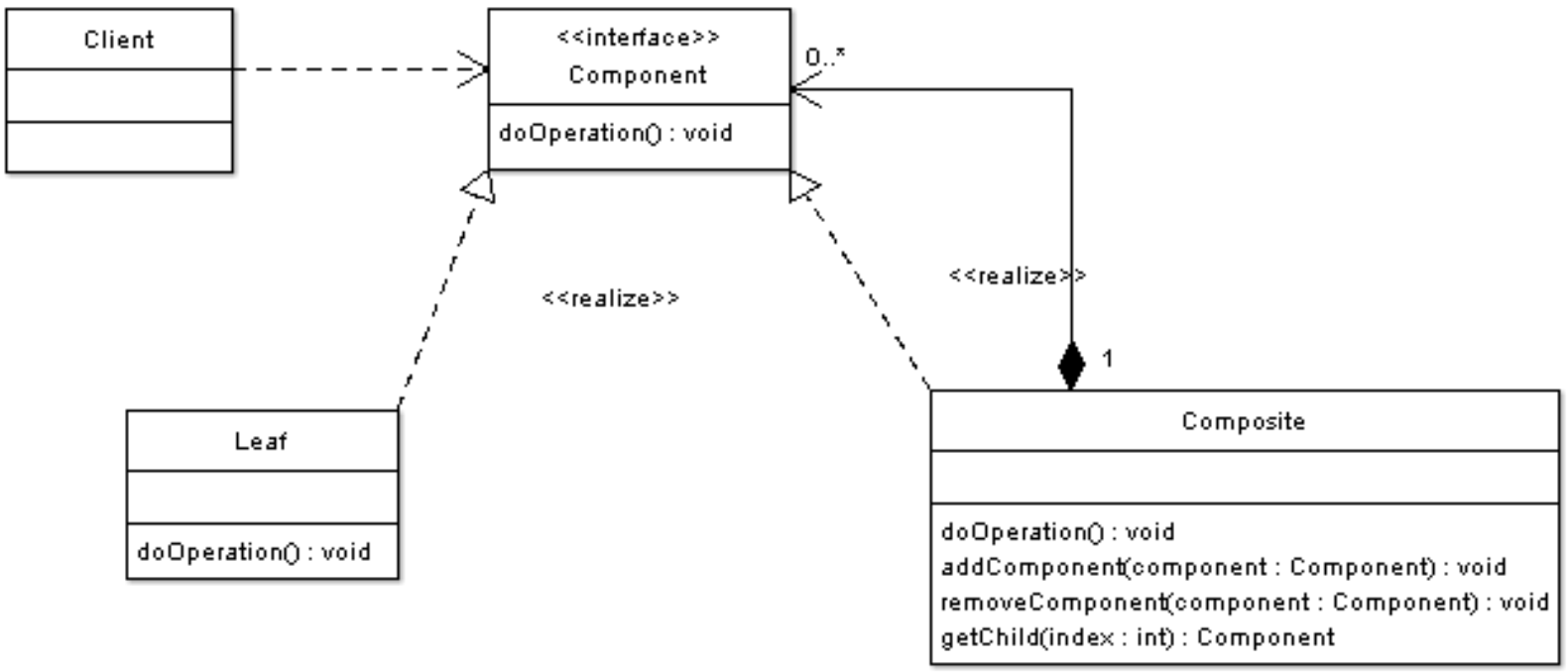
Composite Design Pattern

- **Composite** is a Conceptual design pattern that allows composing objects into a tree-like structure and work with the it as if it was a singular object.
- *Composite pattern lets clients to treat the individual objects in a uniform manner.*
- The Composite Pattern allows you to compose objects into a tree structure to represent the part-whole hierarchy which means you can create a tree of objects that is made of different parts, but that can be treated one big thing.
- Composite lets clients to treat individual objects and compositions of objects uniformly.

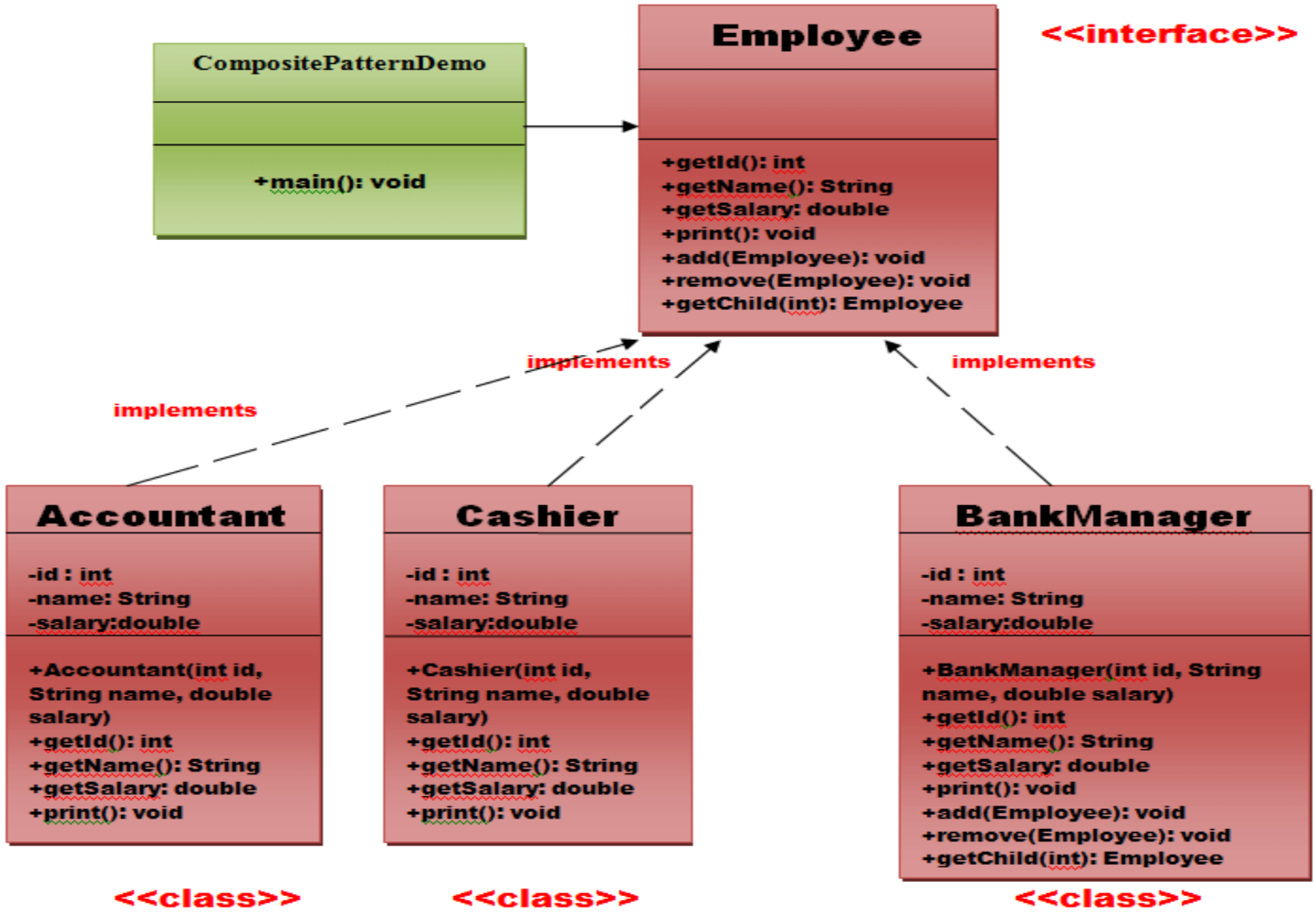
Composite Design Pattern

- For example, a program that manipulates a file system.
- A file system is a tree structure that contains Branches which are Folders as well as Leaf nodes which are Files.
- Note that a folder object usually contains one or more file or folder objects and thus is a complex object where a file is a simple object.
- Note also that since files and folders have many operations and attributes in common, such as moving and copying a file or a folder, listing file or folder attributes such as file name and size.
- It would be easier and more convenient to treat both file and folder objects uniformly by defining a File System Resource Interface

Composite Design Pattern



Composite Design Pattern



Composite Design Pattern

- Composite Pattern important points
- Composite pattern should be applied only when the group of objects should behave as the single object.
- Composite design pattern can be used to create a tree like structure.

Observer Pattern

- In the Observer design pattern, an object (Subject) maintains a list of dependents (Observers) so that the Subject can notify all the Observers about the changes that it undergoes using any of the methods defined by the Observer.
- In the world of distributed applications, multiple services interact with each other to perform a larger operation that a user wants to achieve.
- Services can perform multiple operations, but the operation they perform is directly or heavily dependent on the state of the objects of the service that it interacts with.

Observer Pattern

- Consider a use case for user registration where the user service is responsible for user operations on the website.
- Let's say that we have another service called e-mail service that observes the state of the user and sends e-mails to the user.
- For example, if the user has just signed up, the user service will call a method of the e-mail service that will send an e-mail to the user for account verification.
- If the account is verified but has fewer credits, the e-mail service will monitor the user service and send an e-mail alert for low credits to the user.

Observer Pattern

- Thus, if there's a core service in the application on which many other services are dependent, the core service becomes the Subject that has to be observed/monitored by the Observer for changes.
- The Observer should, in turn, make changes to the state of its own objects or take certain actions based on the changes that happen in the Subject.
- The above scenario, where the dependent service monitor's state changes in the core service, presents a classical case for the Observer design pattern.

Observer Pattern

- Consider the example of a blog. Let's suppose that you're a tech enthusiast who loves to read about the latest articles on Python on this blog.
- You subscribe to the blog. Like you, there would be multiple subscribers that are also registered with the blog. So, whenever there is a new blog, you get notified, or if there is a change on the published blog, you are also made aware of the edits.
- The way in which you're notified of the change can be an e-mail. Now if you apply this scenario to the Observer pattern, the blog is the Subject that maintains the list of subscribers or Observers.
- So when a new entry is added to the blog, all Observers are notified via e-mail or any other notification mechanism as defined by the Observer.

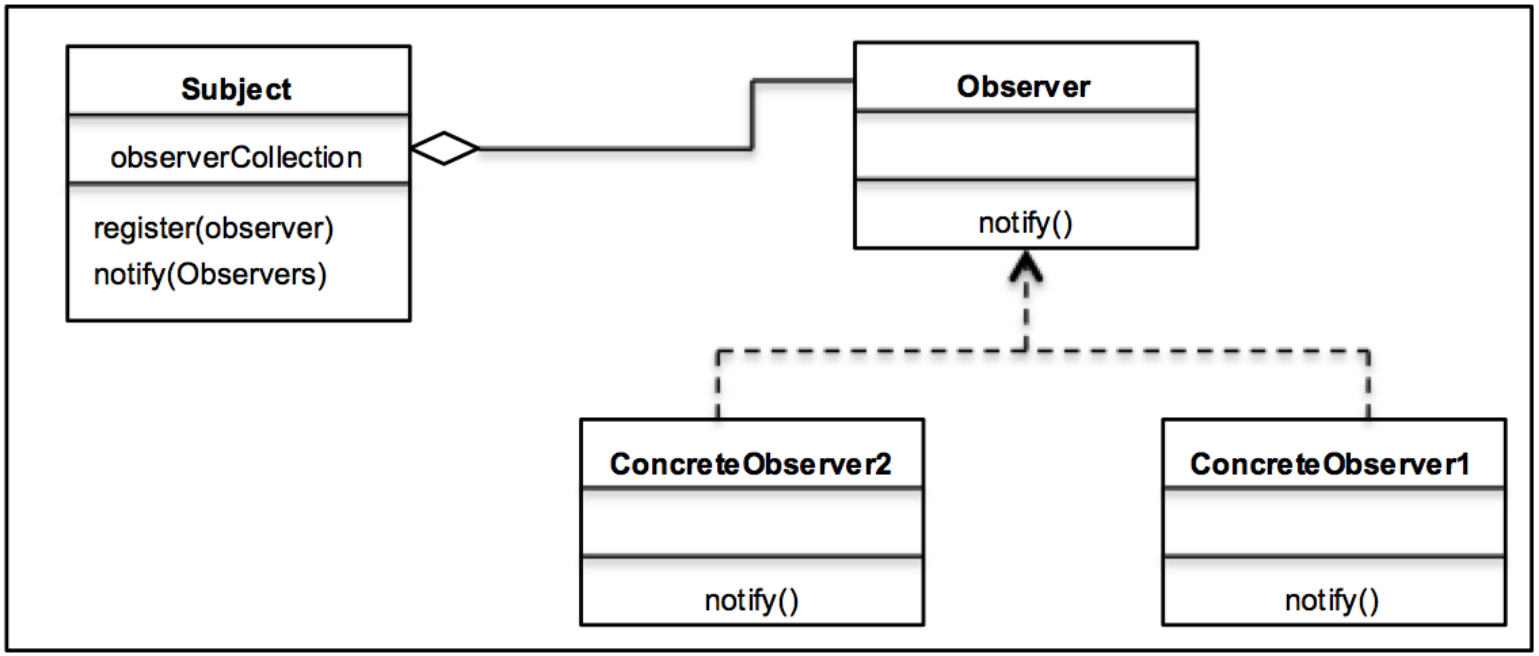
Observer Pattern

- The main intentions of the Observer pattern are as follows:
- It defines a one-to-many dependency between objects so that any change in one object will be notified to the other dependent objects automatically
- It encapsulates the core component of the Subject

Observer Pattern

- The Observer pattern is used in the following multiple scenarios:
- Implementation of the Event service in distributed systems
- A framework for a news agency
- The stock market also represents a great case for the Observer pattern

Observer Pattern



Observer Pattern

- News agencies typically gather news from various locations and publish them to the subscribers. Let's look at the design considerations for this use case.
- With information being sent/received in real time, a news agency should be able to publish the news as soon as possible to its subscribers.
- Additionally, because of the advancements in the technology industry, it's not just the newspapers, but also the subscribers that can be of different types such as an e-mail, mobile, SMS, or voice call. We should also be able to add any other type of subscriber in the future and budgeting for any new technology.

Observer Pattern

- **The Observer pattern methods**
- **The pull model**
- In the pull model, Observers play an active role as follows:
- The Subject broadcasts to all the registered Observers when there is any change
- The Observer is responsible for getting the changes or pulling data from the subscriber when there is an amendment
- The pull model is ineffective as it involves two steps—the first step where the Subject notifies the Observer and the second step where the Observer pulls the required data from the Subject

Observer Pattern

- **The push model**
- In the push model, the Subject is the one that plays a dominant role as follows:
- Unlike the pull model, the changes are pushed by the Subject to the Observer.
- In this model, the Subject can send detailed information to the Observer (even though it may not be needed). This can result in sluggish response times when a large amount of data is sent by the Subject but is never actually used by the Observer.
- Only the required data is sent from the Subject so that the performance is better.

The Observer pattern – advantages and disadvantages

- The Observer pattern provides you with the following advantages:
- It supports the principle of loose coupling between objects that interact with each other
- It allows sending data to other objects effectively without any change in the Subject or Observer classes
- Observers can be added/removed at any point in time

The Observer pattern – advantages and disadvantages

- The Observer interface has to be implemented by ConcreteObserver, which involves inheritance. There is no option for composition, as the Observer interface can be instantiated.
- If not correctly implemented, the Observer can add complexity and lead to inadvertent performance issues.
- In software application, notifications can, at times, be undependable and result in race conditions or inconsistency.

Command Pattern

- The Command design pattern helps us encapsulate an operation (undo, redo, copy, paste, and so forth) as an object.
- What this simply means is that we create a class that contains all the logic and the methods required to implement the operation.

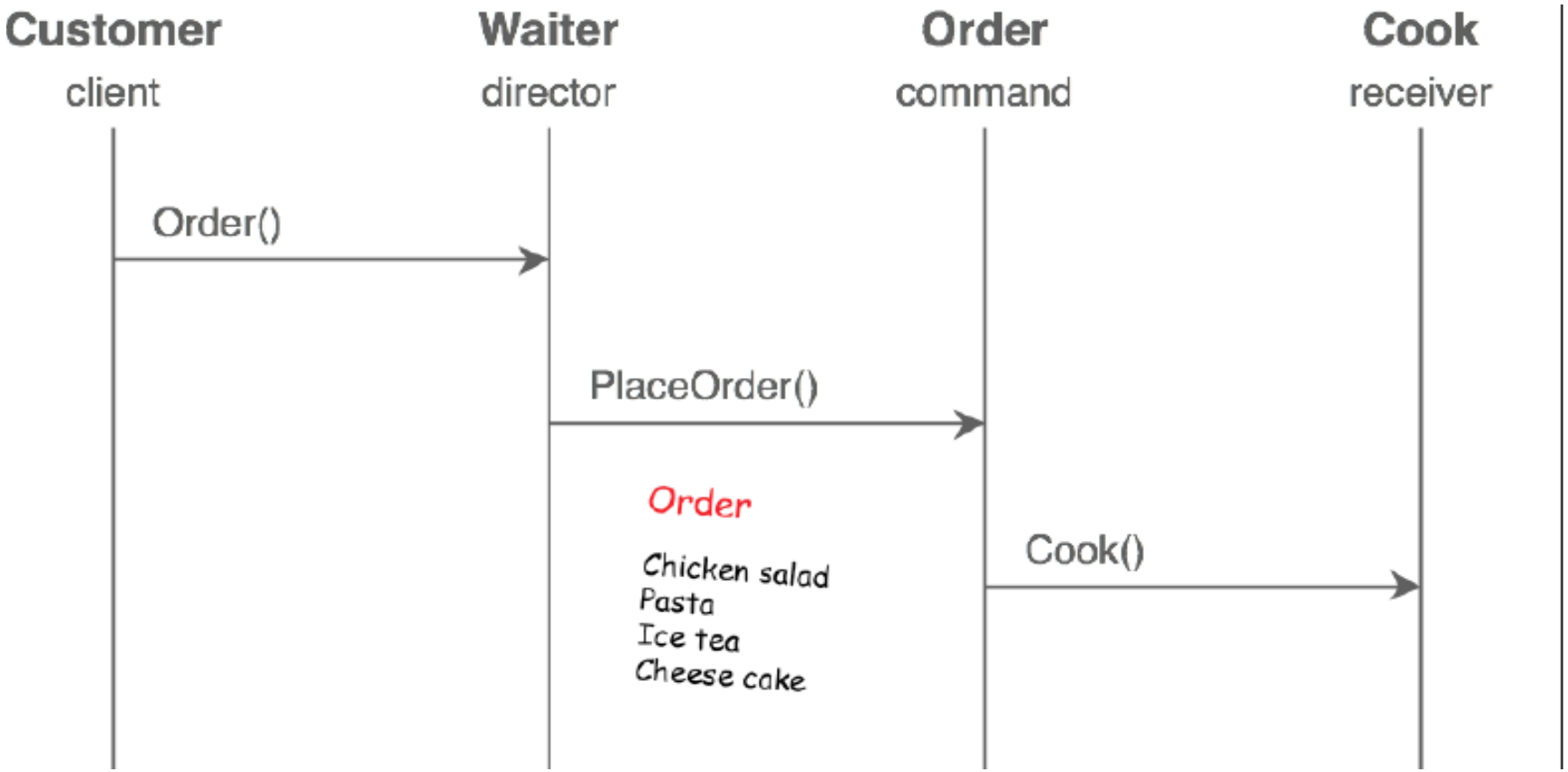
Command Pattern

- We don't have to execute a command directly. It can be executed on will.
- The object that invokes the command is decoupled from the object that knows how to perform it. The invoker does not need to know any implementation details about the command.
- If it makes sense, multiple commands can be grouped to allow the invoker to execute them in order. This is useful, for instance, when implementing a multilevel undo command.

Command Pattern Real Time

- When we go to the restaurant for dinner, we give the order to the waiter.
- The check (usually paper) they use to write the order on is an example of Command.
- After writing the order, the waiter places it in the check queue that is executed by the cook.
- Each check is independent and can be used to execute many and different commands, for example, one command for each item that will be cooked.

Command Pattern Real Time



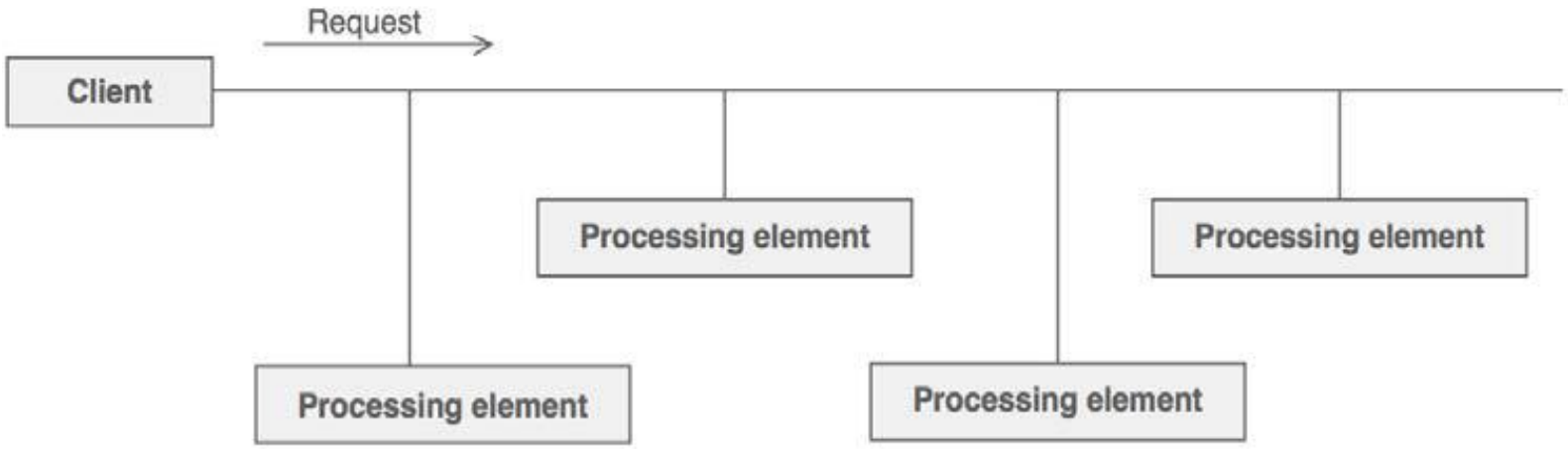
Command Pattern Real Time – Use Case

- **GUI buttons and menu items**: The PyQt example that was already mentioned uses the Command pattern to implement actions on buttons and menu items.
- **Other operations**: Apart from undo, Command can be used to implement any operation. A few examples are cut, copy, paste, redo, and capitalize text.
- **Transactional behavior and logging**: Transactional behavior and logging are important to keep a persistent log of changes. They are used by operating systems to recover from system crashes, relational databases to implement transactions, filesystems to implement snapshots, and installers (wizards) to revert cancelled installations.
- **Macros**: By macros, in this case, we mean a sequence of actions that can be recorded and executed on demand at any point in time. Popular editors such as Emacs and Vim support macros.

The Chain of Responsibility Pattern

- The **Chain of Responsibility** pattern is used when we want to give a chance to multiple objects to satisfy a single request, or when we don't know which object (from a chain of objects) should process a specific request in advance.
- The principle is the same as the following:
- There is a chain (linked list, tree, or any other convenient data structure) of objects.
- We start by sending a request to the first object in the chain.
- The object decides whether it should satisfy the request or not.
- The object forwards the request to the next object.
- This procedure is repeated until we reach the end of the chain.

The Chain of Responsibility Pattern



The Chain of Responsibility Pattern

ATMs and, in general, any kind of machine that accepts/returns banknotes or coins (for example, a snack vending machine) use the chain of responsibility pattern. There is always a single slot for all banknotes, as shown in the following figure

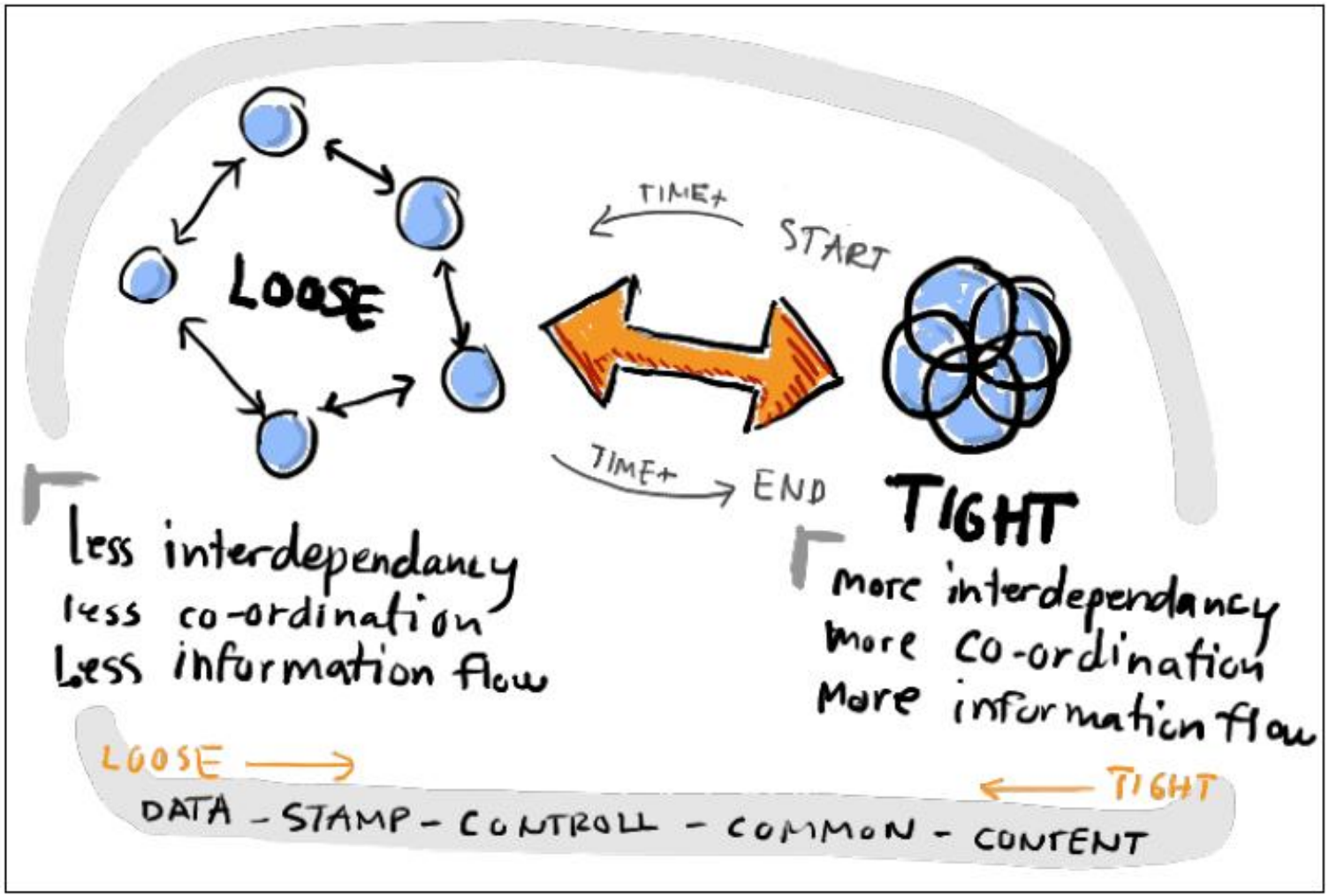
The Chain of Responsibility Pattern usecase

In purchase systems, there are many approval authorities. One approval authority might be able to approve orders up to a certain value, let's say $100.

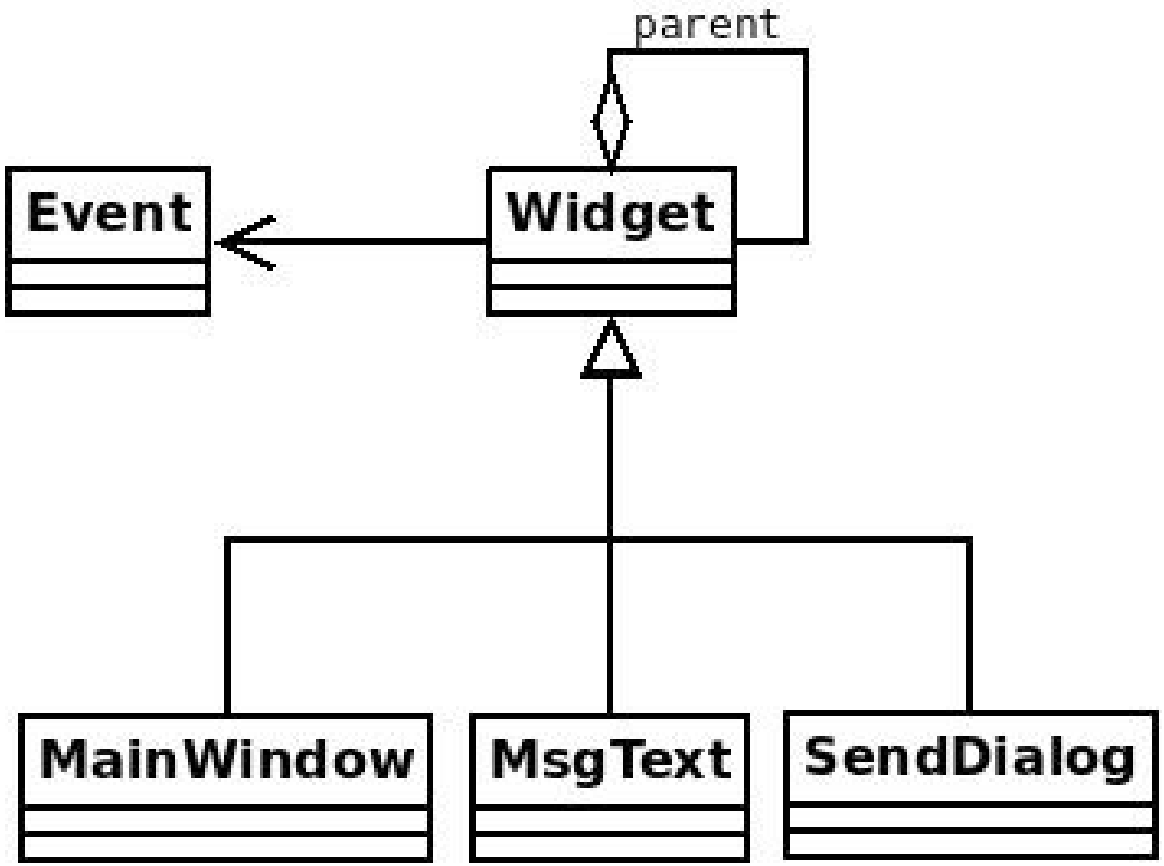
If the order is more than $100, the order is sent to the next approval authority in the chain that can approve orders up to $200, and so forth.

Another case where Chain of Responsibility is useful is when we know that more than one object might need to process a single request. This is what happens in an event-based programming. A single event such as a left mouse click can be caught by more than one listener.

The Chain of Responsibility Pattern usecase



The Chain of Responsibility Pattern usecase



Questions