TypeScript

The Ultimate Guide

zero-to-hero

Disclaimer

This TypeScript series may cause improved coding skills, or a burning desire to dive deeper into TypeScript. Viewer discretion advised. Enjoy responsibly!

Installation

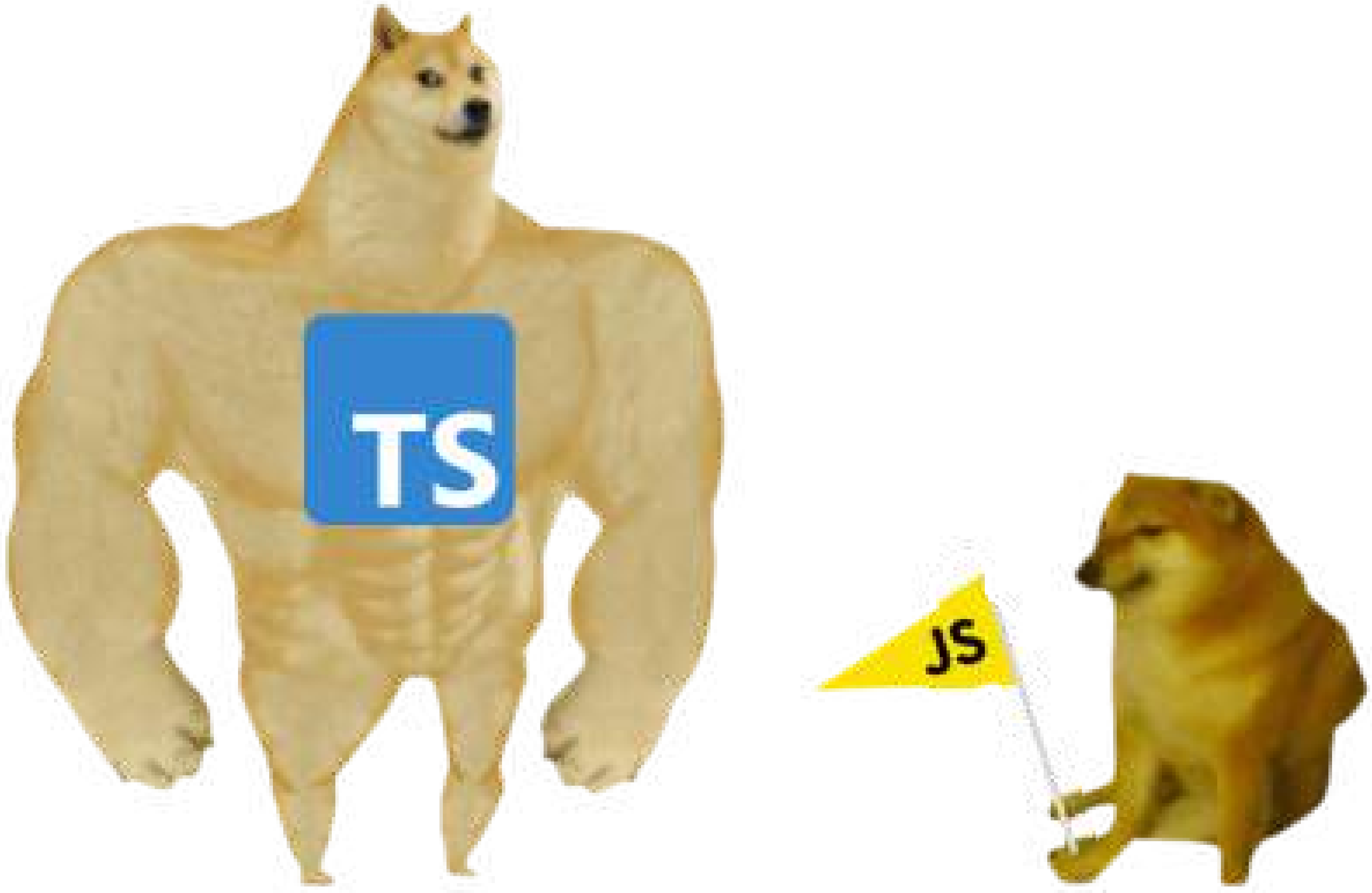
- npm install typescript --save-dev

  Or

- npx tsc

TypeScript

The Basics


TS
JS

Introduction to

TypeScript fundamentals

- TypeScript is a statically-typed superset of JavaScript that compiles to plain JavaScript.
- It provides optional static typing, which enables better tooling and helps catch errors at compile-time.
- This makes our code more scalable and reliable, and we can check that our code runs properly before runtime.

Declaring

Variables

- TypeScript follows the same rules as JavaScript for variable declarations. Variables can be declared using: var, let, and const.
- **var:** Older variable declaration keyword in JavaScript.
- **let:** Introduces block-scoping and declares an immutable variable.
- **const:** Declares an immutable variable and preferred for most cases in TypeScript.

Var

- var declares a variable with function scope or global scope.
- Hoisting: Variables declared with var are moved to the top of their scope during the compilation phase.
- **Example:**

```
console.log(name); // Output: undefined
var name = "Ritik";
```

let

- The let declarations follow the same syntax as var declarations.
- let introduces block-scoping to variables
- **Example:**

```
if (true){
 let count = 10;
 console.log(count); // Output: 10
}
console.log(count); // Error: count is not
// defined
```

Example

```
let name = "Ritik";

let typedName: string = "Ritik";
 // with Type annotations (not necessary)
```

Benefits of

let

- Avoids variable hoisting: Variables declared with let are not hoisted
- Provides better scoping: Variables are confined to the block where they are declared
- Enhances code readability and maintainability

Choosing B/W

var and let

- let is preferable over var.
- let is block-scoped let variables cannot be read or written to before they are declared.
- let variables cannot be re-declared.

const

- const declares an immutable variable that cannot be reassigned.
- Immutable variables are useful for constants or values that should not be changed
- **Example:**

```
const PI = 3.14;
PI = 3.14159; // Error: Cannot assign to 'PI'
// because it is a constant
```

Benefits of

const

- Ensures immutability: Prevents accidental reassignment of values.
- Improves code robustness: Immutable variables can't be modified, reducing potential bugs.
- Expresses intent: Indicates that a variable's value should not change.

Differences B/W

let and const

- let allows variable reassignment, while const does not.
- let can be declared without an initial value, but const requires initialization.
- **Example:**

```
let count: number;
count = 10; // Valid
const PI: number; // Error: Const declarations
// must be initialized
```

Choosing B/W

let and const

- Use let when the value needs to change over time.
- Use const for values that should remain constant
- It is advisable to use const in typescript as it helps improve code clarity and prevents accidental modifications.

Considerations for

const

- const does not make objects or arrays immutable, only their assignment.
- Properties of a const object can be modified.
- **Example:**

```
const person = { name: "Alice", age: 30 };
person.age = 31; // Valid

person = { name: "Bob", age: 40 }; // Error:
 // Cannot assign to 'person' because it is a
 // constant
```

TypeScript

DataTypes

TypeScript
Developer
JAVASCRIPT
JS

Understanding

TypeScript Data Types

- Data types in TypeScript help ensure code reliability, catch errors early, and improve maintainability.
- Data types define the nature of values that variables can hold.
- They specify the operations that can be performed on variables.
- TypeScript is a statically-typed language, allowing explicit declaration of data types.
- const {variable name}: {variable type} = {variable value}

Primitive

data types in TypeScript

- Built-in Data Types: TypeScript provides several primitive types to handle different kinds of data. The most common ones include **number** for numeric values, **string** for textual data, and **boolean** for representing true/false values. Additionally, **null** and **undefined** represent the absence of a value, while **bigint** handles large integers, and **symbol** represents unique identifiers.

- Value-Based Comparison: Primitive types in TypeScript are compared by their value, not by reference. When comparing primitive values using equality operators (=== and !==) to check type also.

Type

annotations and inference

- **Type annotations:** Explicitly declare the type of a variable or function parameter.

  Example:

  ```
  let count: number = 10;
  ```

- **Type inference:** TypeScript automatically infers types based on the assigned value or usage context.

  Example:

  ```
  let message = "Hello, TypeScript!";
  // TypeScript infers message as string
  ```

number

data type in TypeScript

- **number**: Represents numeric values.
- Used to store numbers, both integers and floating-point numbers.
- **Examples**:

```
let age: number = 25;
let pi: number = 3.14;
```

- Supports mathematical operations like addition, subtraction, multiplication, and division.

string

data type in TypeScript

- **string**: Represents textual data
- Used to store sequences of characters.
- **Examples**:

```
let name: string = "John";
let message: string = 'Hello, TypeScript!';
```

- Supports string manipulation, concatenation, and interpolation.

boolean

data type in TypeScript

- **boolean:** Represents true/false values
- Used to store boolean values indicating logical states.
- **Examples**:

```
let isActive: boolean = true;,
let isCompleted: boolean = false;
```

- Supports logical operations like AND (&&), OR (||), and NOT (!).

null and undefined

data type in TypeScript

- **null and undefined:** Represents absence of a value

- null represents the intentional absence of any object value.

- undefined represents the absence of a value for an uninitialized variable.

- **Examples**:

```
let result: null = null;
let data: undefined = undefined;
```

- Often used to indicate the absence of a valid value or an error condition.

Bigint

data type in TypeScript

- **bigint:** Represents arbitrary precision integers
- Used to store integers of arbitrary size.
- **Example**:

```
let bigValue: bigint =
BigInt(1000000000000000);
```

- Enables working with extremely large numbers without losing precision.

Symbol

data type in TypeScript

- **symbol:** Represents unique identifiers
- Used to create unique and immutable values that can be used as object properties or keys.
- **Example**:

```
const id: symbol = Symbol("uniqueId");
```

- Symbols are unique, even if they have the same description.

```
const symbol1 = Symbol("ritik");
const symbol2 = Symbol("ritik");

console.log(symbol1 === symbol2);
 // false
```

Complex

data types in TypeScript

- Complex types in TypeScript, such as arrays, objects, and function types, provide powerful tools for representing structured data, modeling real-world entities, and defining the shape and behavior of functions.

- By leveraging complex types in TypeScript, developers can ensure stronger type safety, enhance code clarity, and enable precise modeling of various data structures and function behaviors.

Arrays

data type in TypeScript

- **Arrays:** represent ordered collections of values.
- **Example**: let numbers: number[] = [1, 2, 3, 4, 5];
- Arrays can store values of the same type, such as numbers or strings, or a combination of different types.
- **Example**: let mixedArray: (number | string)[] = [1, "two", 3, "four"];
- Arrays have built-in methods and properties for manipulation and access, such as push(), pop(), length, and forEach().

Objects

data type in TypeScript

- **Objects:** Represent collections of key-value pairs.
- **Example**: let person: { name: string, age: number } = { name: "John", age: 25 };
- Objects can have properties and methods, allowing you to define and access structured data with meaningful properties and behaviors.
- **Example**: person.name = "Jane";

Objects

data type in TypeScript

- Objects can be nested within other objects to create complex data structures.

```
let user: {
  name: string,
  address: {
    street: string,
    city: string
  }
} = {
  name: "John",
  address: {
    street: "123 Main St",
    city: "New York"
  }
};
```

Functions

and types in TypeScript

- Functions in TypeScript can have parameter types and return types for type safety.

```
function addNumbers(a: number, b: number): number {
  returna+b;
}
```

- Functions can be assigned to variables and passed as arguments to other functions.

```
const calculate = addNumbers;

console.log(calculate(5, 10)); // Output: 15
```

Functions

and types in TypeScript

- Functions can also have optional parameters using "?" and default parameter values.

```
function greet(name: string, greeting?: string): void {

 if (greeting) {
 console.log(`${greeting}, ${name}!`);
 } else {
console.log(`Hello, ${name}!`);
 }
}

greet("John"); // Output: Hello, John!
greet("Jane", "Good morning");
// Output: Good morning, Jane!
```

Union

types in TypeScript

- **Union types (|):** Combine multiple types, allowing a value to have different possible types.

- **Example:**

```
let value: string | number = "Hello";
```

© Ritik Banger

TypeScript

“type” keyword

- Creating Custom Types: The "type" keyword allows creating type aliases for custom types.

  Example: type Age = number;

- The type keyword helps in annotating and reusing union types.

  Example:

```
type Status = "active" | "inactive";
let iAmActive: Status = "active";
let iAmInactive: Status = "inactive";
```

Intersection

types in TypeScript

- **Intersection types (&):** Combine multiple types, requiring a value to have all the specified types.

- **Example:**

```
type Person = { name: string; age: number; };
type Employee = { id: number; department: string; };
type EmployeeDetails = Person & Employee;

const employee: EmployeeDetails = {
name: "John Doe",
age: 30,
id: 123,
department: "HR",
 };
```

Special

types in TypeScript

- TypeScript has special types that may not refer to any specific type of data.
- These special types are any, unknown, and, never.
- Other special types are void, null, and, undefined

any

- The any type is a special type in TypeScript that represents a value of any type.
- It allows variables to bypass type checking and enables dynamic behavior.
- Use the any type sparingly, as it sacrifices type safety and can lead to potential runtime errors.
- **Example:**

```
let data: any = 10;
data = "Hello";
data = true;
```

TypeScript code
as any

unknown

- The **unknown** type represents a value whose type is unknown at compile-time.
- Variables of type unknown can hold values of any type but require type checking or assertions for safe usage.
- It provides a safer alternative to the any type when dealing with dynamic or uncertain data.
- **Example:**

```
let value: unknown = "Hello";
let length: number;
if (typeof value === "string") {
  length = value.length;
} else {
  length = 0;
}
```

© Ritik Banger

never

- The **never** type represents a type that never occurs.
- It is used to indicate values that cannot happen or functions that never return.
- It is particularly useful in exhaustiveness checking and enforcing strict handling of all possible cases.
- Functions returning never must either throw an error or have an infinite loop.
- **Example:**

```
function throwError(): never {
  throw new Error("An error occurred");
}
throwError();  // Function that throws an
error and never returns.
```

void

- The **void** type represents the absence of any type in TypeScript.

- It is commonly used as the return type of functions that do not explicitly return a value.
- Variables of type void can only be assigned undefined or null.

- Functions returning never must either throw an error or have an infinite loop.

- **Example:**

```
function logMessage(message: string): void {
console.log(message);
}
logMessage("Hello, TypeScript!");
 // Prints "Hello, TypeScript!"
```

TypeScript

Interfaces

Javascript variable
Random data types
Typescript
Random data types

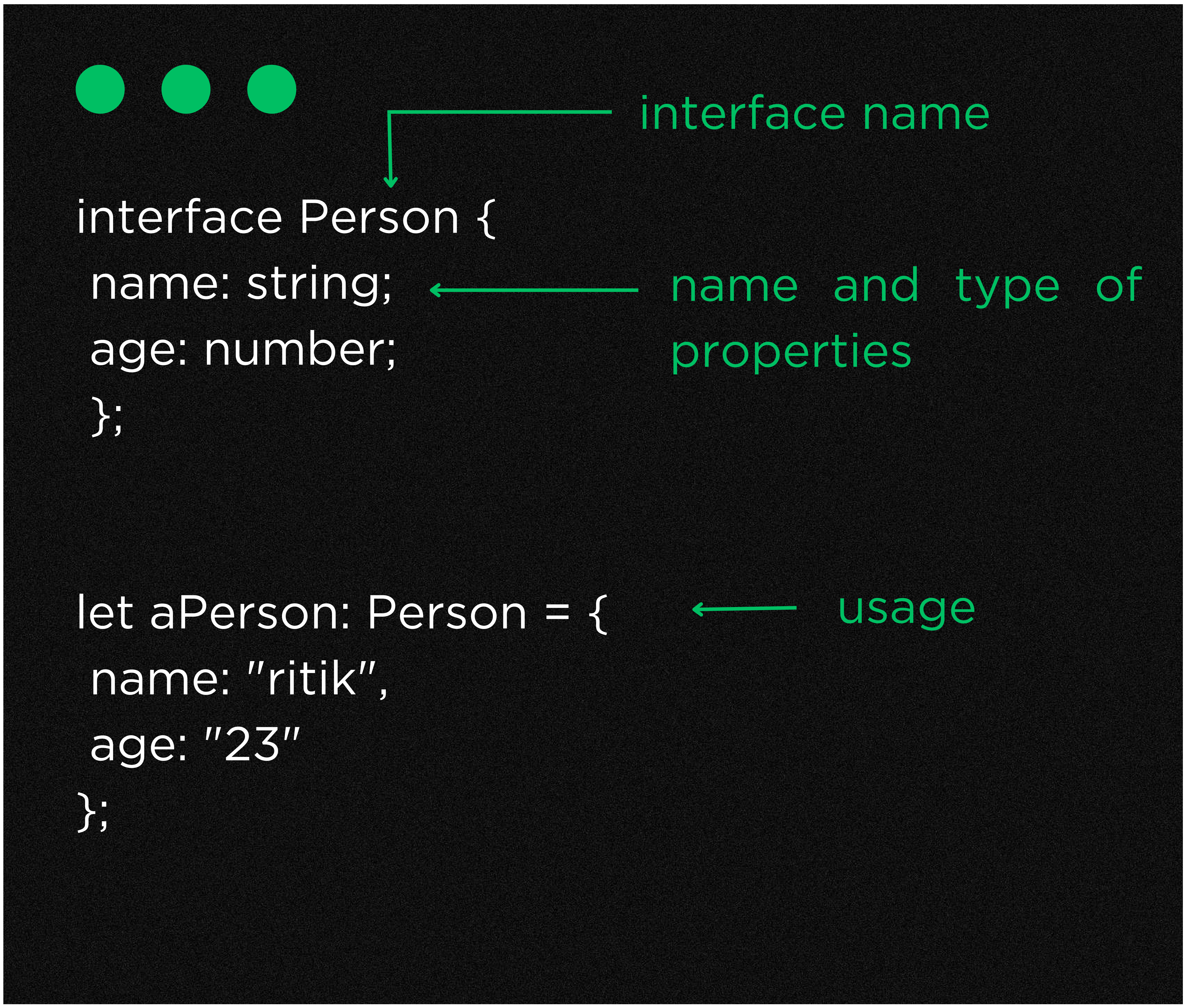
Understanding

TypeScript Interfaces

- TypeScript interfaces provide a powerful way to define object structures and ensure type safety.
- Interfaces define the structure of an object by specifying the names and types of its properties.
- We declare an interface using the interface keyword in a .ts file. You can also create interface inside your component file with .tsx extension.

interface

in TypeScript



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


let aPerson: Person = {
 name: "ritik",
 age: "23"
};
```

Working

with TypeScript interface

- Interfaces support contract-based programming, enabling clear communication and agreement on the expected structure and behavior of objects in TypeScript code.

- Interfaces provide a way to achieve abstraction and polymorphism, allowing for the creation of reusable and interchangeable components that adhere to a common interface

Optional

Properties

- Interfaces can have optional properties denoted by a "?" after the property name.

```
interface Book {
 title: string;
 author: string;
 year?: number;
};

const validBook: Book = {
 title: "typescript learning",
 author: "ritik"
};
```

Optional

Properties

- If you provide with properties that does not exist in interface, you will get an error like this

```
const invalidBook: Book = {
 title: 'Invalid Book',
 author: 'Unknown',
 meta: 'Some additional information',
};

/* Type '{ title: string; author: string;
meta: string; }' is not assignable to type
'Book'. Object literal may only specify
known properties, and 'meta' does not
exist in type 'Book'. */ };
```

Readonly

Properties

- Interfaces can have readonly properties that cannot be modified after initialization.

```
interface Point {
readonly x: number;
readonly y: number;
};
 const point: Point = {
 x: 5,
 y: 10,
};

// Attempting to modify a readonly property

point.x = 8; // Error: Cannot assign to 'x' because it is a
 // read-only property.
```

Function

Types

- Interfaces can define the shape of a function, including parameter types and return types.

```
interface MathOperation {
(x: number, y: number): number;
};

const add: MathOperation = (x, y) => x + y;
const subtract: MathOperation = (x, y) => x - y;

const result1 = add(5, 3); // result1 = 8
const result2 = subtract(10, 4); // result2 = 6
```

Extending

Interfaces

- Interfaces can extend other interfaces, inheriting their properties and adding new ones.

```
interface Shape {
 color: string;
};

 interface Square extends Shape {
 sideLength: number;
};

const square: Square = {
 color: 'red',
 sideLength: 5,
};
```

Implementing

Interfaces

- Classes can implement interfaces, ensuring they adhere to the defined structure.

```
interface Printable {
print(): void;
};

class Document implements Printable {
 print() {
   console.log("Printing document...");
  };
};

const doc = new Document();
doc.print(); // "Printing document..."
```

Generic

Interfaces

- Interfaces in TypeScript can also be generic, allowing for flexibility and reusability.

- They enable the creation of interfaces that can work with different types.

- By leveraging generic interfaces, you can write versatile and type-safe code that accommodates a wide range of data types.

Example:

```
interface Box<T> {
  value: T;
};

// Usage with a specific type
const stringBox: Box<string> = {
  value: "Hello, world!",
};

console.log(stringBox.value);  //  Output:  Hello,
//world!

// Usage with another type
const numberBox: Box<number> = {
  value: 42,
};

console.log(numberBox.value); // Output: 42
```

Explanation:

- In this example, we have a generic interface called **Box<T>**, which represents a simple box that can hold a value of any type T. The value property within the Box interface is of type T.
- We create an instance **stringBox** of type **Box<string>** and assign the value "Hello, world!" to it.
- The value stored in **stringBox** is accessed using the value property, which we log to the console, resulting in the output "Hello, world!".
- Similarly, we create an instance **numberBox** of type **Box<number>** and assign the value 42 to it.
- We access the value stored in **numberBox** using the value property, which we log to the console, resulting in the output 42.

Differences

Types v/s Interfaces

- Type aliases and interfaces are very similar, and in many cases you can choose between them freely. Almost all features of an interface are available in type.

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

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

Types v/s Interfaces

- Interfaces support declaration merging, allowing you to define multiple interface declarations with the same name, and they are merged into a single interface with the combined properties while Types do not support declaration merging.

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

const person: Person = {
  name: 'John Doe',
  age: 30,
};
```

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

// Error: Duplicate
//identifier 'Person'.
// in case of type
```

TypeScript

type assertion, const assertion, and type aliases

TYPESCRIPT
BEST HAI RE BABA

Type Assertion

in typescript

- Type assertion allows you to override the inferred type of a variable.
- Type assertions do not have any runtime impact and are purely for compile-time type checking.
- They provide hints to the compiler on how we want our code to be analyzed.
- They allow us to override the inferred type and explicitly specify the desired type.

Type Assertion

in typescript

- Type assertions are used when we have more knowledge about a value's type than TypeScript can infer.
- Type assertions are particularly useful when working with union types or dynamically typed values.
- TypeScript provides two ways to do Type Assertion.

  Using **Angular Bracket (<>)**
  Using "**as**" keyword

Type assertions are, simply put, evil.

They are meant to tell the compiler: "I know what I am doing, and I know it better than you".

Unless there is a really good reason, do not use type assertions.

Type Assertion

Using Angular Bracket (<>)

- Angular bracket syntax allows you to assert the type of a variable.
- Syntax: **<TypeName>variableName**
- Example:

```
let myVariable: any = 'Hello';
let length: number = (<string>myVariable).length;
console.log(length) //5
```

Type Assertion

Using "as" keyword

- The "as" keyword is another way to assert the type of a variable. It is
- Syntax: **variableName as TypeName**
- Example:

```
let myVariable: string = '100';
let fiveHundred: number = (myVariable as number) * 5;
console.log(fiveHundred) //500
```

Differences

Why "as" over "<>"

- Angular Bracket (<>) is the older syntax, while "as" keyword is the newer syntax introduced in TypeScript 1.6.

- The "as" keyword can be used with type arguments for type narrowing, while Angular Bracket syntax does not support this.

- It is recommended to use the "as" keyword for type assertion in modern TypeScript projects.

Const Assertion

in typescript

- Const assertion is a feature in TypeScript that allows you to specify that a variable should have a literal type.
- It provides a way to enforce immutability and treat values as constant throughout the program.
- Syntax: **variableName as const**
- Example

```
let myArray = [1, 2, 3] as const;
myArray.push(4); // Error: Cannot push to readonly
// array
```

Benefit

of const assertion

- Provides stronger guarantees about immutability and constness of values.
- Enables TypeScript to narrow down the types of values more accurately.
- Prevents accidental modification of values, leading to safer code.
- Combine const assertions with readonly modifiers for enhanced immutability.

Usage

const assertion

- Example 1: Creating an array of literal values:

```
let daysOfWeek = ['Sunday', 'Monday', 'Tuesday',
'Wednesday', 'Thursday', 'Friday', 'Saturday'] as
const;
```

- Example 2: Defining an object with constant properties:

```
let person = {
 name: 'Ritik',
 gender: 'male',
} as const;
```

Limitation

const assertion

- The constness is only enforced at compile-time; it doesn't affect the runtime behavior.
- Nested properties of objects and arrays are not automatically inferred as const.
- Const assertions cannot be used with variables that have union types.

Type Aliases

in typescript

- Type aliases allow you to create custom names for types, improving code readability and reusability.
- They provide a way to define a type once and reuse it throughout our codebase.
- Type aliases are declared using the type keyword followed by the alias name and the type definition.
- Example:

```
type Age = number;
```

Usage

Type Aliases

- Type aliases can be used to simplify complex types or provide more meaningful names.

```
type Point = {
  x: number;
  y: number;
};

const onePoint: Point = {
 x:10,
 y:5
};
```

Usage

Type Aliases- Union Types

- Type aliases can be used with union types to create more expressive types for variables.

- Example

```
type userId = string | number;

const userOne: userId = "100";

const userTwo: userId = 100;
```

Example:

```
type Shape = Circle | Square | Triangle;

type Circle = {
  kind: 'circle';
  radius: number;
};

type Square = {
  kind: 'square';
  sideLength: number;
};

type Triangle = {
  kind: 'triangle';
  base: number;
  height: number;
};

let myShape: Shape = { kind: 'circle', radius: 5 };
```

Usage

Type Aliases- Intersection Types

- Type aliases can also be used with intersection types to combine multiple types into one for variables.
- & is used between two type aliases to create a new intersection type.
- intersection types in TypeScript are typically used to combine object types and cannot be created with primitive types alone. Intersection types are designed to merge the properties of multiple object types into a single type that encompasses all the combined properties.

Example:

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

type Employee = {
  employeeId: string;
  department: string;
};

type EmployeeInfo = Person & Employee;

let employee: EmployeeInfo = {
  name: 'Ritik',
  age: 23,
  employeeId: 'E123',
  department: 'Engineering',
};
```

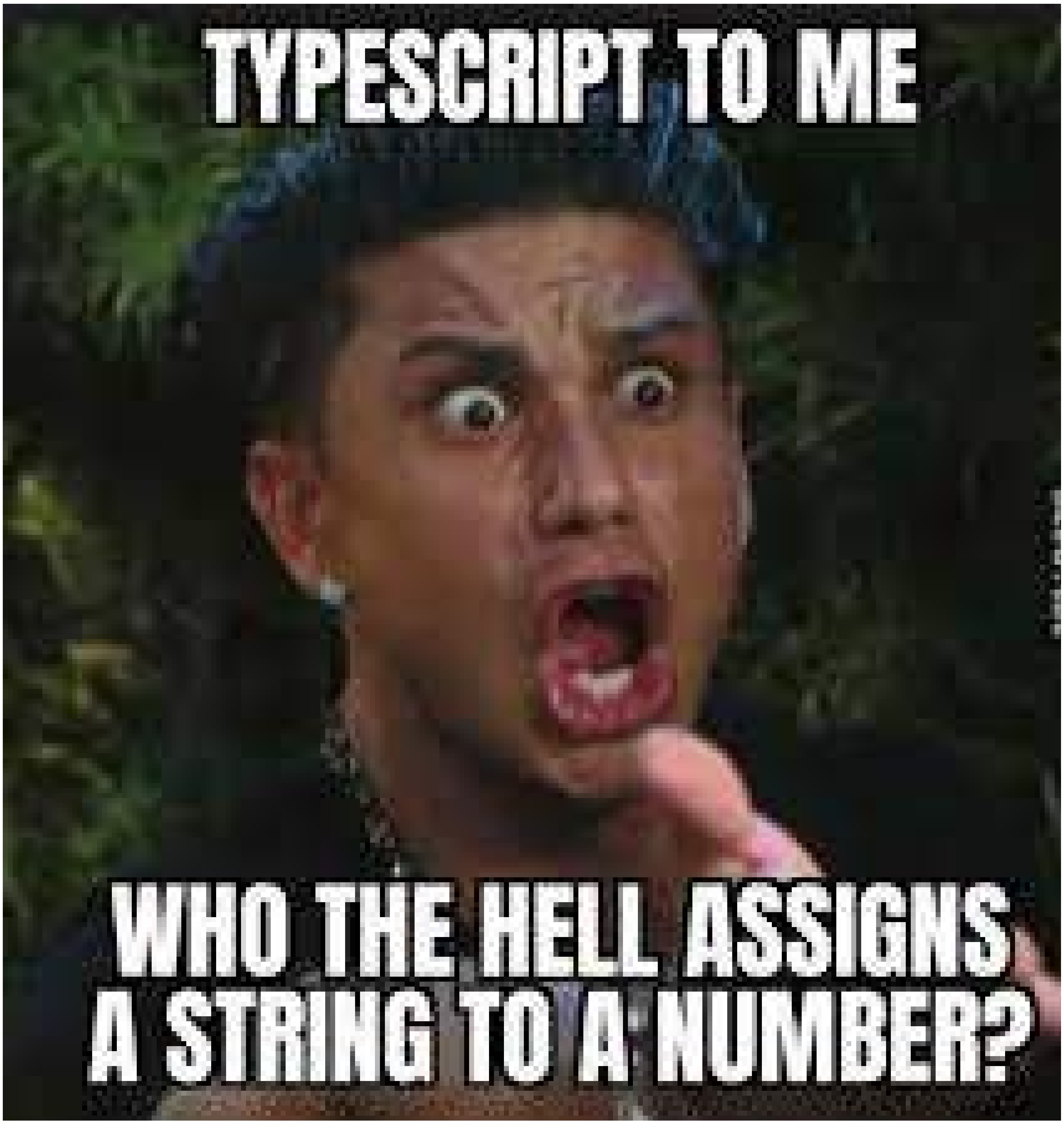
Benefits

Type Aliases

- Improved code readability and understandability.
- Reduced repetition and code duplication.
- Simplified complex types and enhanced type expressiveness.
- Easier maintenance and refactoring.

TypeScript

Index Types


TYPESCRIPT TO ME
WHO THE HELL ASSIGNS A STRING TO A NUMBER?

Introduction

Index Types

- Index Types enable dynamic property access in TypeScript.
- They provide a flexible way to define and access properties on objects.
- There are two concepts for index types- Index operators, and Index signatures

Index operators

in index types

- There are two concepts in Index Operators- Index type query operator: keyof, and Indexed access operator: T[K].
- The keyof operator takes an object type and produces a string or numeric literal union of its keys.
- We can use an indexed access operator (T[K]) to look up a specific property on another type

Keyof Operator

Index type query operator

- The keyof operator allows us to obtain the keys of a given type.
- The key can be any string, and the value can be of a specific type or a union of types.
- Example:

```
interface Person{
name: string;
age: number
};
type PersonKey = keyof Person;
// type PersonKey = "name" | "age"
```

T[K]

Indexed access operator- Lookup Types

- Lookup Types enable property lookups based on key types.

- Lookup types ensure type safety when accessing properties. By utilizing lookup types, TypeScript can infer and enforce the correct types of properties retrieved through lookup operations. This helps prevent runtime errors by ensuring that the accessed properties exist and have the expected types.

Example:

```
interface Person {
  name: string;
  age: number;
}
type PersonInfo = {
  name: string;
  age: number;
};
type PersonLookup = {
  [key: string]: PersonInfo;
};
const people: PersonLookup = {
  john: { name: "John Doe", age: 25 },
  jane: { name: "Jane Smith", age: 30 },
};

function getPersonInfo(key: keyof PersonLookup): PersonInfo |
undefined {
  return people[key];
}

const johnInfo = getPersonInfo("john");
console.log(johnInfo); // Output: { name: "John Doe", age: 25 }

const janeInfo = getPersonInfo("jane");
console.log(janeInfo); // Output: { name: "Jane Smith", age: 30 }
```

Explanation:

In this example, we have an interface Person representing the properties of a person. We also define a type **PersonInfo** that captures a subset of properties from **Person** (name and age).

Next, we define a type **PersonLookup** using a lookup type. The **PersonLookup** type allows for dynamic keys of type string and values of type **PersonInfo**. This enables us to create an indexed collection of person information.

We initialize the **people** object as an instance of **PersonLookup**, where each key represents a person's identifier ("john", "jane"), and the corresponding value is the person's information.

The **getPersonInfo** function takes a key parameter of type keyof PersonLookup and retrieves the corresponding PersonInfo from the people object. The return type is PersonInfo or undefined if the key is not found.

We call the **getPersonInfo** function with different keys ("john", "jane") and store the returned person information in variables (johnInfo, janeInfo).

Index signatures

in index types

- It can be used to specify structure of object with arbitrary property names.
- The syntax of an index signature is simple and looks similar to the syntax of a property. But with one difference: write the type of the key inside the square brackets:

{ [key: KeyType]: ValueType }.

Example:

```
interface Person {
  name: string;
  age: number;
  [key: string]: string | number;
}

const person: Person = {
  name: "John",
  age: 30,
  city: "New York",
};
```

Best Practices

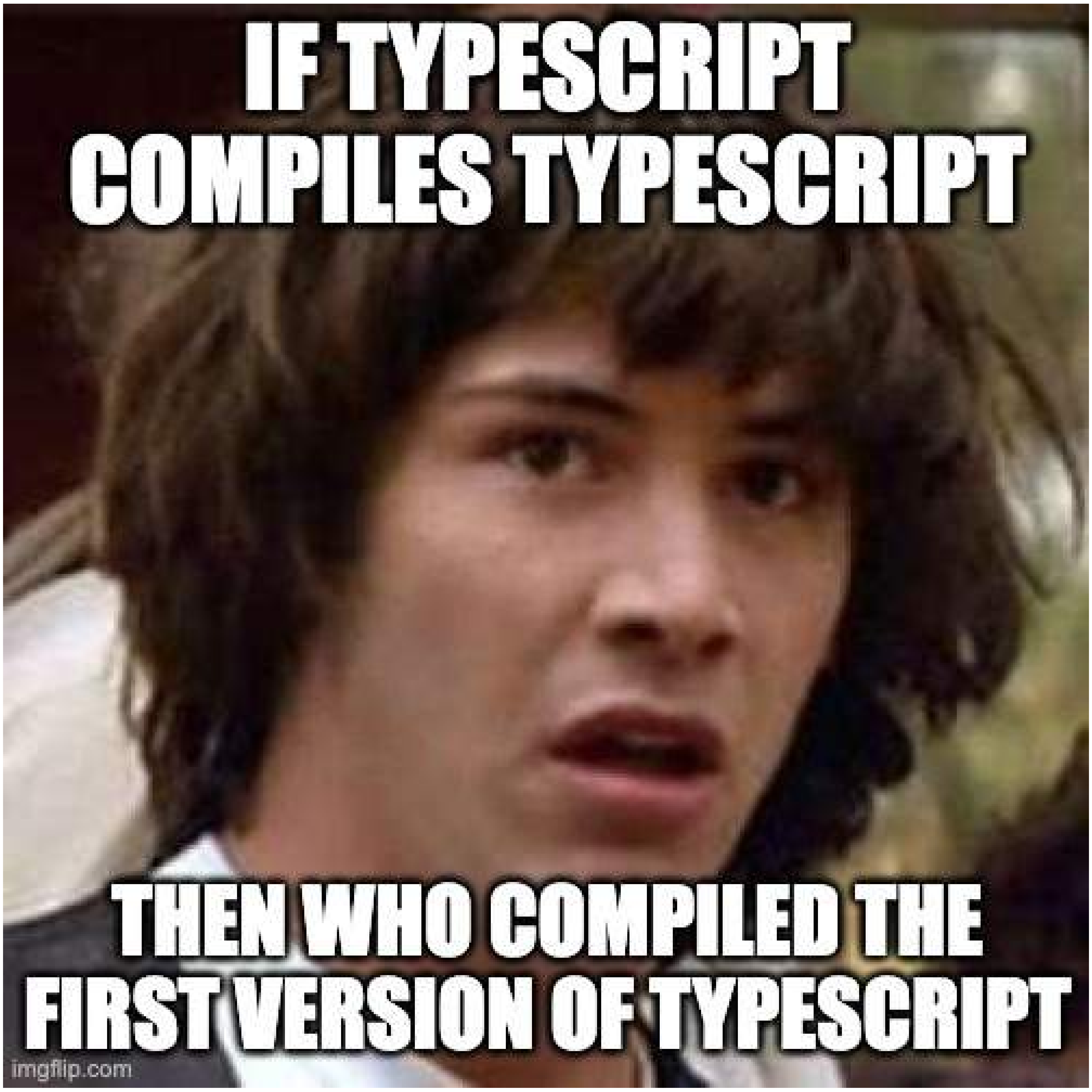
Index Types

- Avoid overly generic index types to maintain type safety.
- Consider using mapped types and utility types like Pick and Omit for more fine-grained control over index types.

(we will cover this in next week)

TypeScript

Mapped Types and Conditional Types


IF TYPESCRIPT COMPILES TYPESCRIPT
THEN WHO COMPILED THE FIRST VERSION OF TYPESCRIPT
imgflip.com

Introduction

Mapped Types

- Mapped types allow you to transform and manipulate existing types.
- Mapped types are new types based on other types
- Mapped types build on the syntax for index signatures, which are used to declare the types of properties which have not been declared ahead of time.
- Example: **{ readonly [P in keyof T]: T[P] }** adds "readonly" to all properties of type T.

Mapped Types

Let's understand with an example

- Without Mapped Types

```
// Configuration values for the current user
type AppConfig = {
 username: string;
 layout: string;
};

// Whether or not the user has permission
// to change configuration values

type AppPermissions = {
 changeUsername: boolean;
 changeLayout: boolean;
};
```

Example

Mapped Types

- This example is problematic because there is an implicit relationship between **AppConfig** and **AppPermissions**. Whenever a new configuration value is added to **AppConfig**, there must also be a corresponding boolean value in **AppPermissions**.

- It is better to have the type system manage this relationship than to rely on the discipline of future program editors to make the appropriate updates to both types simultaneously.

Mapped Types

Let's understand with an example

- With Mapped Types

```
// Configuration values for the current user
type AppConfig = {
 username: string;
 layout: string;
};

// Whether or not the user has permission to change
 //  configuration values
type AppPermissions = {
[Property in keyof AppConfig as
`change${Capitalize<Property>}`]: boolean
};

//Usage
let permission: AppPermissions = {
changeLayout: true,
changeUsername: false
};
```

© Ritik Banger

Key Features

Mapped Types

- Enumerate and iterate over the properties of an existing type.
- Modify properties and their types.
- Create new properties with modified types.

TypeScript

Conditional Types

- Conditional types enable you to make type decisions based on conditions.
- Useful for creating type-safe functions and utility types.
- Conditional types let us deterministically define type transformations depending on a condition.
- They are a ternary conditional operator applied at the type level rather than at the value level.
- Example: **T extends U ? X : Y** selects type X if T is assignable to U, otherwise selects type Y.

Example:

```
type NonNullable<T> = T extends null |
undefined ? never : T;

type NullableString = string | null;
type NonNullableString =
NonNullable<NullableString>;

const str: NonNullableString = "Hello";
const nullableStr: NonNullableString = null;
// Error: Type 'null' is not assignable to type
 // 'string'.
```

Conditional Types

Let's understand with an example

- Without Conditional Types

```
type ExtractIdType<T extends {id: string | number}> = T["id"];

interface NumericId {
 id: number
};
interface StringId {
 id: string
};
interface BooleanId {
 id: boolean
};

type NumericIdType = ExtractIdType<NumericId>
// type NumericIdType = number
type StringIdType = ExtractIdType<StringId>
 // type StringIdType string
type BooleanIdType=ExtractIdType<BooleanId>
// won't work
```

- With Conditional Types- By simply moving the constraint in the conditional type, we were able to make the definition of BooleanIdType work.

```
type ExtractIdType<T> = T extends {id: string | number} ?
T["id"] : never;

interface NumericId {
 id: number
};
interface StringId {
 id: string
};
interface BooleanId {
 id: boolean
};

type NumericIdType = ExtractIdType<NumericId>
// type NumericIdType = number
type StringIdType = ExtractIdType<StringId>
// type StringIdType = string
type Boolean IdType = ExtractIdType<BooleanId>
 // type BooleanIdType = never
```

- In this case, we refined the ExtractIdType type. Instead of forcing the type of the id property to be of type string | number, we've introduced a new type U using the infer keyword. Hence, BooleanIdType won't evaluate to never anymore. In fact, TypeScript will extract boolean as expected.

```
type ExtractIdType<T> = T extends {id: infer U} ?
T["id"] : never;

interface BooleanId {
 id: boolean
};

type BooleanIdType = ExtractIdType<BooleanId>
// type BooleanIdType = boolean
```

Key Features

Conditional Types

- Type branching based on conditions.
- Inference of types based on conditions.
- Combination with union and intersection types.

Distributive conditional types

- Distributive conditional types automatically distribute over union types.
- Example: **T extends U ? X : Y** applied to a union type **A | B** will distribute the condition over A and B.

```
type ExtractStrings<T> = T extends string ? T : never;

type StringOrNumber = string | number;

type ExtractedStrings = ExtractStrings<StringOrNumber>;

// ExtractedStrings: string
```

Inbuilt conditional types

- TypeScript provides built-in conditional types for common type manipulations.
- Examples: **Exclude<T, U>**, **Extract<T, U>**, **NonNullable<T>**, **ReturnType<T>**, etc.
- NonNullable<T> filters out the null and undefined values from a type T

```
type NonNullable<T> = T extends null | undefined ? never : T

type A = NonNullable<number> // number
type B = NonNullable<number | null> // number
type C = NonNullable<number | undefined> // number
type D = NonNullable<null | undefined> // never
```

Inbuilt conditional types

- Parameters<T> and ReturnType<T> let us extract all the parameter types and the return type of a function type, respectively:

```
type Parameters<T> = T extends (...args: infer P) => any ?
P: never

type ReturnType<T> T extends (...args: any) => infer R ? R
: any

type A = Parameters<(n: number, s: string) => void>
 // [n: number, s: string]
type B = ReturnType<(n: number, s: string) => void>
// void
type C = Parameters<() => () => void> // []
type D = ReturnType<() => () => void> // () => void
type E = ReturnType<D> // void
```

Inbuilt conditional types

- ConstructorParameters<T> and InstanceType<T> are the same things as Parameters<T> and ReturnType<T>, applied to constructor function types rather than to function types.
- There are so many utlilty types that are based on conditional types, we will explore them in the next week.

Best Practices

Mapped Types and Conditional Types

- Use mapped types and conditional types to increase type safety and code flexibility.
- Utilize mapped types to create reusable utility types.
- Consider performance implications when using complex mapped types or conditional types.

Performance Considerations

Mapped Types and Conditional Types

- Complex mapped types and conditional types can impact compilation and runtime performance.
- Avoid deeply nested transformations or recursive conditional types when not necessary.
- Profile and optimize if performance becomes a concern

Real-World Use Cases:

Mapped Types and Conditional Types

- Generating immutable versions of existing data structures.
- Creating utility types for data validation or transformation.
- Adapting API responses to a standardized format.

TypeScript

Utility Types, Generics and Global TypeScript

TYPESCRIPT DEVELOPERS

Introduction

Utility Types

- Utility types in TypeScript are powerful tools that enable us to manipulate and transform types.

- They come built-in with TypeScript and provide useful functionalities to enhance our development experience.

- TypeScript provides several utility types to facilitate common type transformations. These utilities are available globally.

Partial<Type>

Utility Types

- The Partial utility type allows us to make all properties of a given type optional. It's perfect when dealing with forms or API payloads where certain fields are optional.

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

type PartialUser = Partial<User>;
/* Result:
{
  name?: string;
  email?: string;
  age?: number;
}
*/
```

Required<Type>

Utility Types

- The Required utility type is the opposite of Partial. It makes all properties of a given type mandatory, ensuring they cannot be undefined.

```
interface Product {
  name?: string;
  price?: number;
}

type RequiredProduct = Required<Product>;
// Result: { name: string; price: number; }
```

Readonly<Type>

Utility Types

- The Readonly utility type allows us to create an immutable version of a type. Once applied, the properties of the type become read-only and cannot be modified.

```
interface Configuration {
  apiUrl: string;
  apiKey: string;
}

type ReadonlyConfiguration = Readonly<Configuration>;
/* Result:
{
  readonly apiUrl: string;
  readonly apiKey: string;
}
*/
```

Record<Key, Type>

Utility Types

- The Record utility type helps us create a new type by mapping specific keys to a particular value type. It's ideal for creating dictionaries or lookup tables.

```
type Fruit = "apple" | "banana" | "orange";
type Price = number;

type FruitPrices = Record<Fruit, Price>;
/* Result:
{
  apple: number;
  banana: number;
  orange: number;
}
*/
```

Pick<Type, Keys>

Utility Types

- The Pick utility type enables us to create a new type by selecting only the specified keys from an existing type. It comes in handy when we need to work with a subset of properties.

```
interface Employee {
  id: number;
  name: string;
  department: string;
  role: string;
}

type EmployeeNameAndRole = Pick<Employee, "name" | "role">;
/* Result:
{
  name: string;
  role: string;
}
*/
```

Omit<Type, Keys>

Utility Types

- Conversely, the Omit utility type allows us to exclude specific keys from an existing type, providing a type without those properties.

```
interface Car {
  brand: string;
  model: string;
  color: string;
}

type CarWithoutColor = Omit<Car, "color">;
/* Result:
{
  brand: string;
  model: string;
}
*/
```

Extract<Type, Union>

Utility Types

- The Extract utility type extracts from a union type (Type) those types that are assignable to another given type (Union). This helps us manipulate union types more effectively.

```
type Animal = "cat" | "dog" | "bird";
type Pet = "cat" | "dog";

type MyPet = Extract<Animal, Pet>;
// Result: "cat" | "dog"
```

Exclude<Type, ExcludedUnion>

Utility Types

- The Exclude utility type creates a new type by excluding all the types from a union (Type) that are assignable to another union (ExcludedUnion). This allows us to filter out unwanted types.

```
type Color = "red" | "blue" | "green";
type PrimaryColors = "red" | "blue";

type SecondaryColors = Exclude<Color, PrimaryColors>;
// Result: "green"
```

NonNullable<Type>

Utility Types

- The NonNullable utility type helps us create a type by removing null and undefined from a given type. This ensures that the resulting type cannot have null or undefined as its values.

```
type NullableString = string | null | undefined;

type NonNullableString = NonNullable<NullableString>;
// Result: string
```

ReturnType<Function>

Utility Types

- The ReturnType utility type allows us to obtain the return type of a function type. This is useful for building generic functions that adapt to the return type of the functions they work with.

```
function greet(): string {
  return "Hello, TypeScript!";
}

type Greeting = ReturnType<typeof greet>;
// Result: string
```

Generics

in TypeScript

- Generics in TypeScript allow us to create flexible and reusable components that work with different data types. They enable writing functions and classes that can handle various data types without sacrificing type safety.

```
function echo<T>(value: T): T {
  return value;
}

const result1 = echo<string>("Hello, TypeScript!");
const result2 = echo<number>(42);
```

Generics

in TypeScript

- The letter **T** is a type parameter, also known as a type variable, used to denote a generic type. When we define a function with a type parameter, it means the function can accept arguments of any data type, and TypeScript will infer and maintain the type of the argument and the return value based on the type of T.

- When we call the **echo** function and specify a type argument, the value passed as an argument must match that type. For example, echo<string>("Hello, TypeScript!") means we are calling the echo function with the type argument string, so the input and output must both be of type string.

Generics

in TypeScript

- Generic Function with Array

```
function reverseArray<T>(array: T[]): T[] {
  return array.reverse();
}

const numbers = [1, 2, 3, 4, 5];
const reversedNumbers = reverseArray<number>(numbers);

const fruits = ["apple", "banana", "orange"];
const reversedFruits = reverseArray<string>(fruits);
```

Generics

in TypeScript

- Generic Interface

```
interface Pair<T, U> {
  first: T;
  second: U;
}

const numberPair: Pair<number, string> = { first: 42, second:
"forty-two" };

const stringPair: Pair<string, boolean> = { first: "hello",
second: true };
```

Generics

in TypeScript

- Generic Class

```
class Box<T> {
  private value: T;

  constructor(initialValue: T) {
    this.value = initialValue;
  }

  getValue(): T {
    return this.value;
  }
}

const numberBox = new Box<number>(42);
const stringValue = numberBox.getValue();
```

Generics

in TypeScript

- Constraints with Generics

```
interface Lengthy {
  length: number;
}

function printLength<T extends Lengthy>(value: T): void {
  console.log(`Length: ${value.length}`);
}

printLength("Hello, TypeScript!"); // Length: 19
printLength([1, 2, 3, 4, 5]); // Length: 5
```

Introduction

Global TypeScript Declarations (global.d.ts)

- The global.d.ts file in TypeScript allows us to declare global variables, functions, or types that are accessible throughout our entire TypeScript project. It acts as a central place to define custom global declarations and provides better type checking for external libraries and environment-specific variables.

- You can name the file anything with an extension of "**.d.ts**", examples are type.d.ts, global.d.ts

Global Types

Why Use global.d.ts?

- Declare Global Variables: When using external libraries or global objects like window or document, global.d.ts lets us define their types, ensuring type safety across the project.
- Augmenting Existing Types: We can extend or modify existing TypeScript types to match our specific requirements within the global.d.ts file.
- Simplify Imports: With global declarations, we no longer need to import certain types explicitly in multiple files, reducing redundancy in our codebase.

Usage

global.d.ts

- Create a global.d.ts file in the root of your TypeScript project.
- Here we are extending Request interface to include userId also.

```
declare namespace Express {
  interface Request {
    userId: string; // Add any other properties related to the
 // user if needed
  }
}

export {}
//exporting the file so that it can be infered as  module
```

Usage

global.d.ts

- Assume we have a simple authentication middleware that sets the userId property in the request object after verifying the user's token.

```
import { Request, Response, NextFunction } from 'express';

export const authMiddleware = (req: Request, res: Response, next:
NextFunction) => {
  // Assume there's a function to validate the user's token and
// retrieve the user ID
  const userId = getUserIdFromToken(req.headers.authorization);

  if (userId) {
    req.userId = userId;
    next();
  } else {
    return res.status(401).json({ message: 'Unauthorized' });
  }
};
```

Usage

global.d.ts

- userService.ts

```
import { Request } from 'express';

class UserService {

  static getUserInfo(req: Request) {
    // Access the userId property from the request object
    const userId = req.userId;

    // Now you can use the userId to fetch user data from the
//database or perform other operations
    // For example:
    return getUserDataFromDatabase(userId);
  }
}
```

Usage

global.d.ts

- userController.ts

```
import { Request, Response } from 'express';
import { UserService } from './userService';

class UserController {
  static getUserInfo(req: Request, res: Response) {
    try {
      const userData = UserService.getUserInfo(req);
      return res.status(200).json(userData);
    } catch (error) {
      return res.status(500).json({ message: 'Something went wrong'
});
    }
  }
}

export default UserController;
```

Setting tsconfig.json

for global.d.ts

- Open your tsconfig.json file and add the "files" property to include the global.d.ts file explicitly. If you already have an "include" property, you can add the path to global.d.ts there.

```
{
  "compilerOptions": {
    "target": "es6",
    "module": "commonjs",
    "strict": true,
    // Add other compiler options as needed

    // Add the "files" property to include the global.d.ts file
    "files": ["global.d.ts"]
  },
  // Add other configuration settings as needed
}
```

Don't forget to

Like, Comment, and Repost