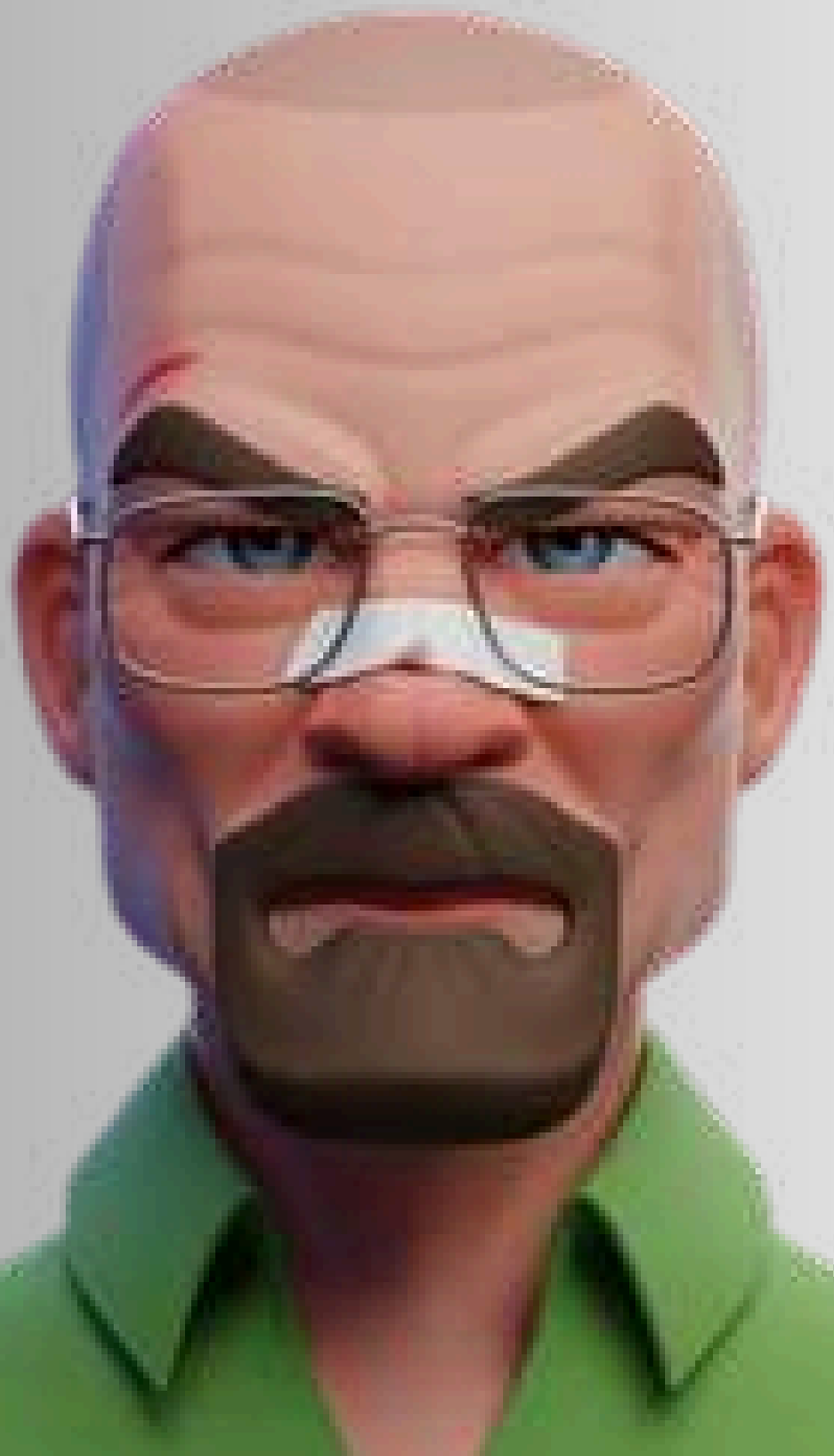
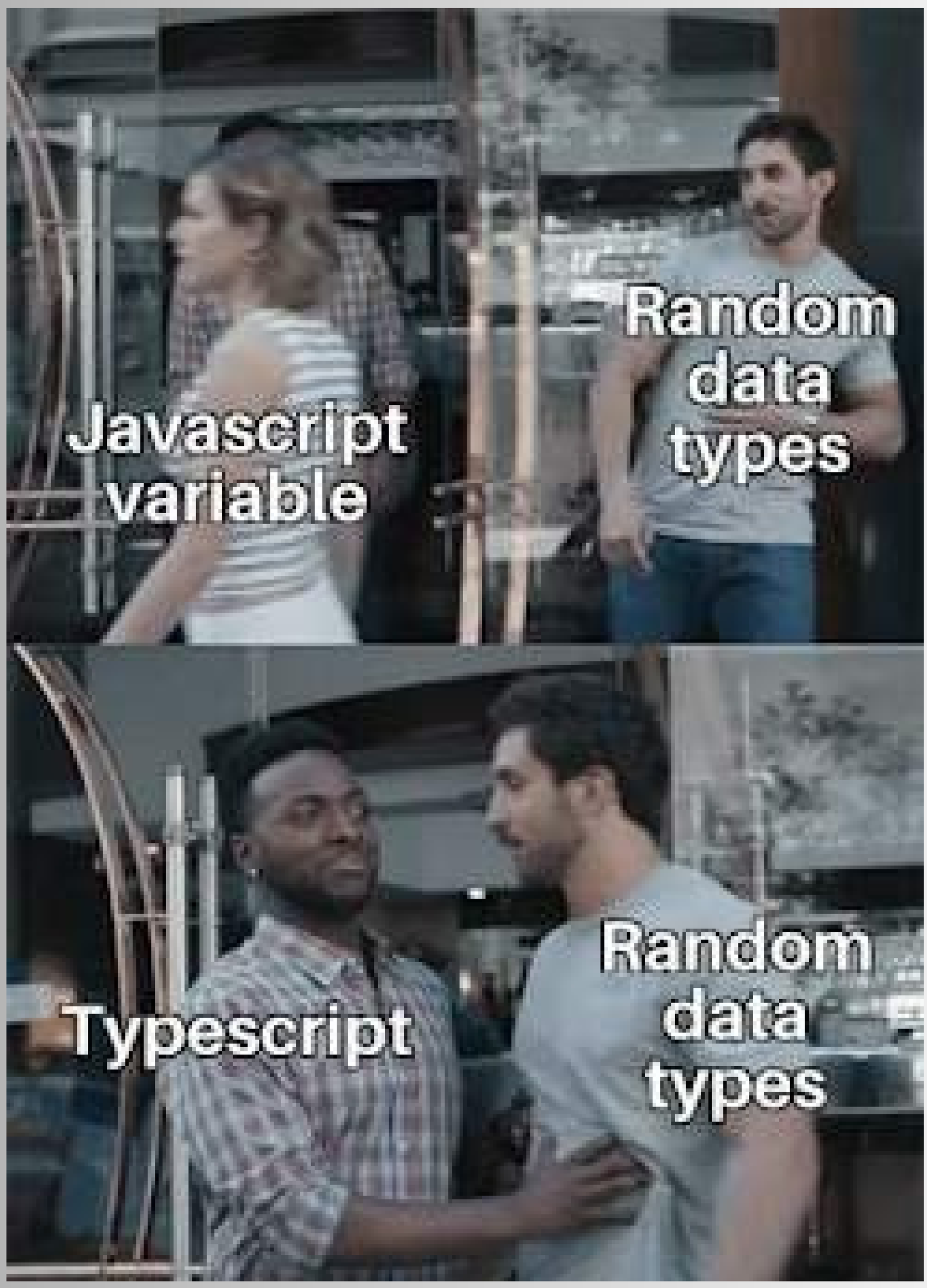
TypeScript

Interfaces



@codewithrafaqat


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;
 };
```

interface name

name and type of properties

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

usage

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

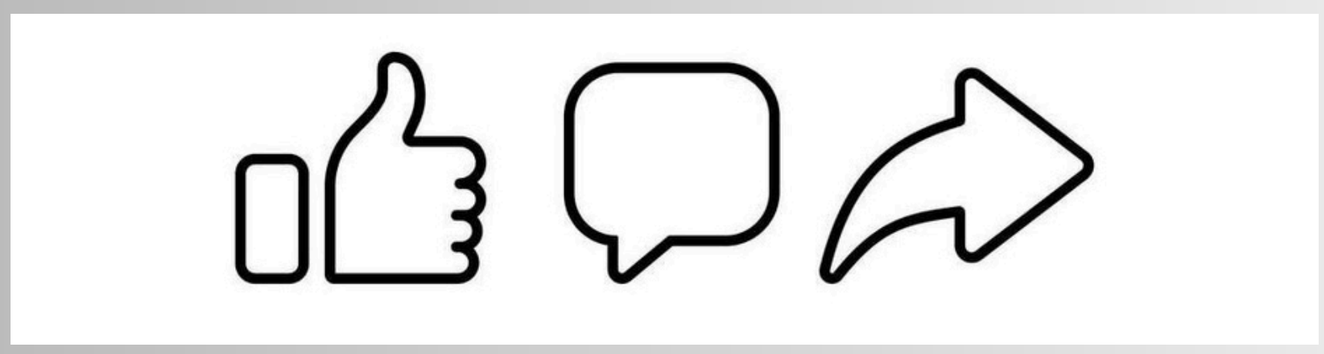
- Interfaces support declaration merging, allowing you to define multiple interface declarations with the same name, and they are merged into a single interface with the combined properties while Types do not support declaration merging.

```
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
```

Don't forget to

Like, Comment, and Repost



@codewithrafaqat