One Guide For TypeScript Mastery

Mohammad Faisal

@mohammadfaisalkhan

Follow for
Web Development & Programming

Save

Installing and running TypeScript

Enter **sudo npm i -g typescript** in the Terminal. This installs Typescript compiler globally using npm.

Enter **tsc --init** to initialize a Typescript project with the **tsconfig.json** file.

Enter **tsc filename.ts** to run a Typescript file.

Mohammad Faisal
@mohammadfaisalkhan

Save

Modifying the **tsconfig.json** file

1. **rootDir** — Specify the root dir of TS files which should be compiled.
2. **outDir** — Specify the dir where the compiled JS files will be exported.
3. **removeComments** — Removes comments after compilation.
4. **noImplicitAny** — Specifies whether TS compiler should give warning for Implicit declaration of 'any' type. (For eg. in functions, where the parameter can be of 'any' type). Set it to false to disable the warnings.
5. **noUnusedParameters** — Show warning for unused function parameters.
6. **noUnusedLocals** — Show warning for unused local variables.
7. **noImplicitReturns** — Specify whether or not to show warnings for implicit returns.

Mohammad Faisal
@mohammadfaisalkhan

Save

Data Types

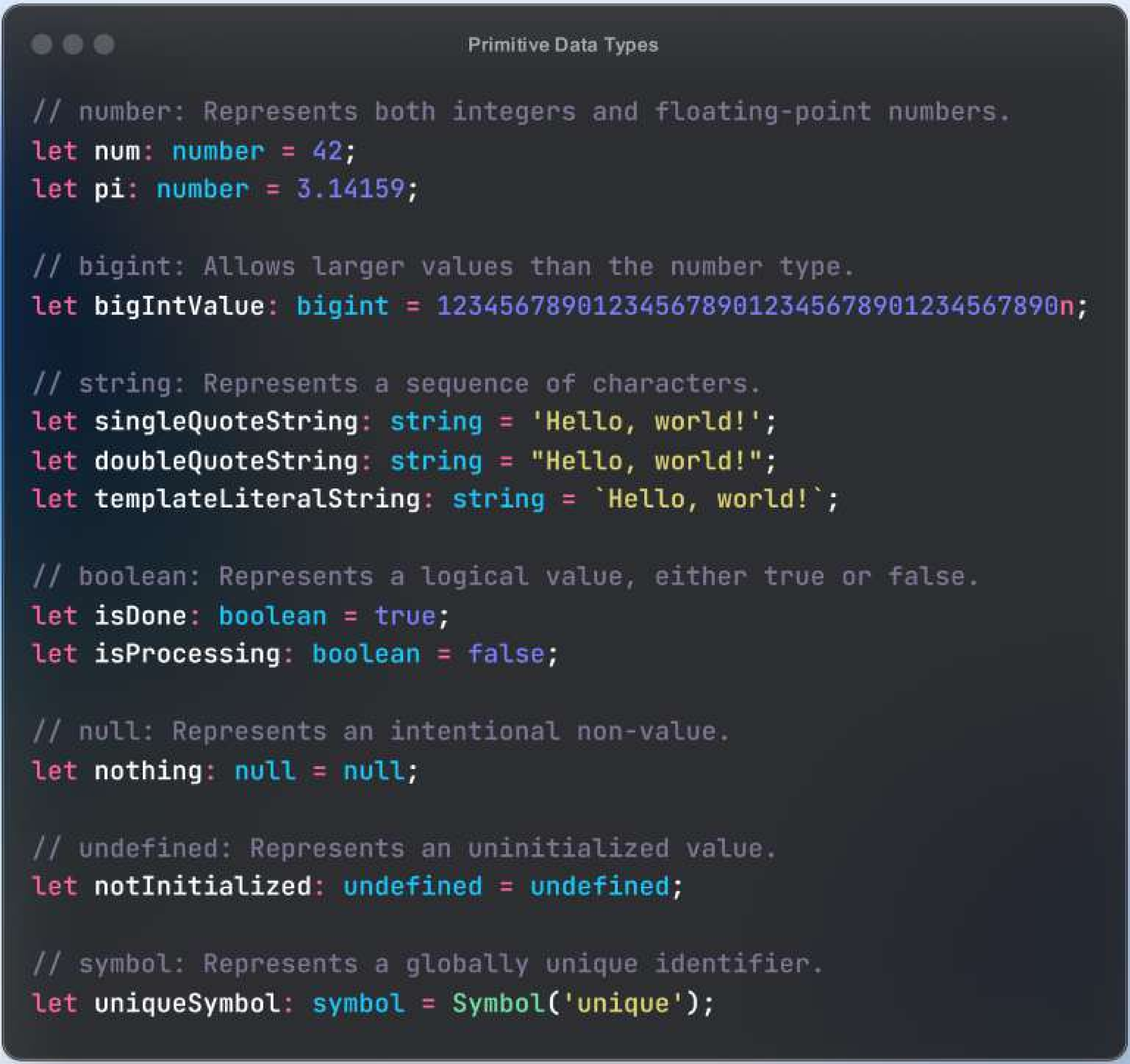
Primitive Data Type

1. **number**: Represents both integers and floating-point numbers.
2. **bigint**: Represents whole numbers and floating point values, but allows larger negative and positive numbers than the number type.
3. **string**: Represents a sequence of characters, enclosed in single quotes ('), double quotes ("), or backticks (`).
4. **boolean**: Represents a logical value, either true or false.
5. **null**: Represents an intentional non-value.
6. **undefined**: Represents an uninitialized value.
7. **symbol**: Represents a globally unique identifier.

Mohammad Faisal
@mohammadfaisalkhan

Save

Primitive Data Types Examples



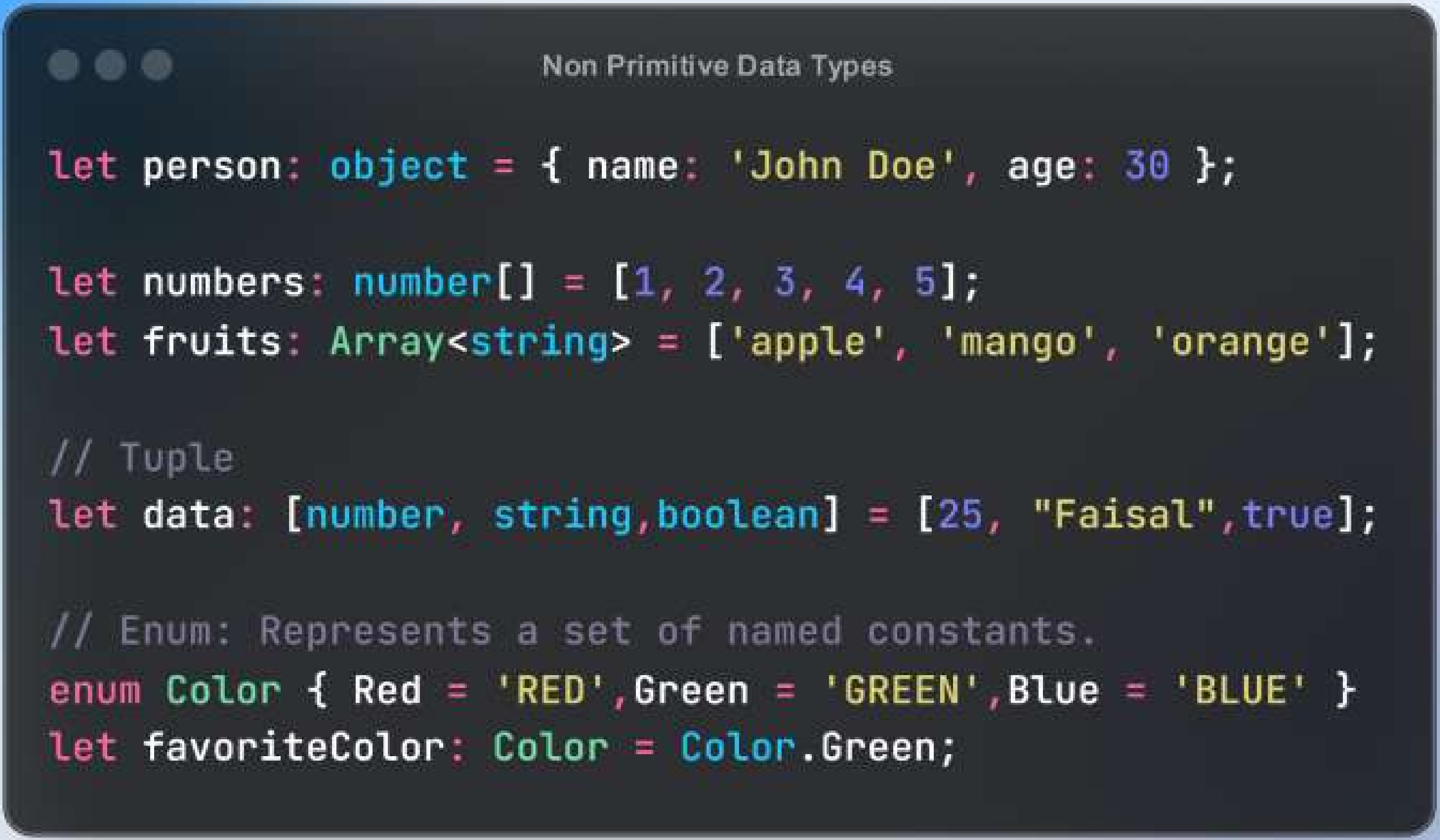
```
// number: Represents both integers and floating-point numbers.
let num: number = 42;
let pi: number = 3.14159;

// bigint: Allows larger values than the number type.
let bigIntValue: bigint = 1234567890123456789012345678901234567890n;

// string: Represents a sequence of characters.
let singleQuoteString: string = 'Hello, world!';
let doubleQuoteString: string = "Hello, world!";
let templateLiteralString: string = `Hello, world!`;

// boolean: Represents a logical value, either true or false.
let isDone: boolean = true;
let isProcessing: boolean = false;

// null: Represents an intentional non-value.
let nothing: null = null;

// undefined: Represents an uninitialized value.
let notInitialized: undefined = undefined;

// symbol: Represents a globally unique identifier.
let uniqueSymbol: symbol = Symbol('unique');
```

Mohammad Faisal
@mohammadfaisalkhan

Save

Non-Primitive Data Types

1. **object**: It can be an instance of a class, an object literal, an array, or another complex data structure.
2. **Array**: Represents a collection of values of the same type. You can define an array using the array literal syntax [] or the generic Array<T> syntax.
3. **Tuple**: Represents an array with a fixed number of elements, where each element can have a different type.
4. **Enum**: Represents a set of named constants.


Non Primitive Data Types

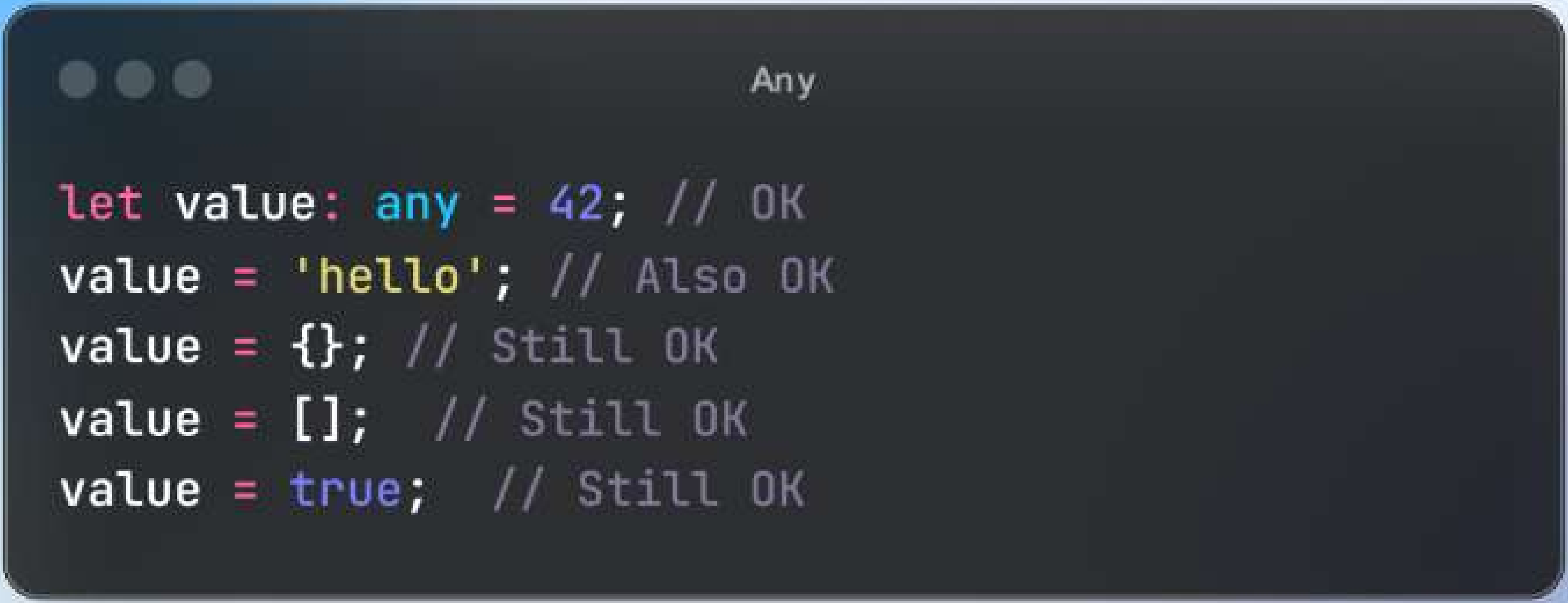
```
let person: object = { name: 'John Doe', age: 30 };

let numbers: number[] = [1, 2, 3, 4, 5];
let fruits: Array<string> = ['apple', 'mango', 'orange'];

// Tuple
let data: [number, string,boolean] = [25, "Faisal",true];

// Enum: Represents a set of named constants.
enum Color { Red = 'RED',Green = 'GREEN',Blue = 'BLUE' }
let favoriteColor: Color = Color.Green;
```

Mohammad Faisal
@mohammadfaisalkhan

Save

Any Data Type

The **any** type in TypeScript allows you to opt out of type-checking, essentially bypassing TypeScript's type system. When a variable is of type `any`, TypeScript does not enforce type safety or checks, which can lead to potential runtime errors and bugs. While it provides flexibility, relying on `any` defeats the purpose of TypeScript's strong typing and can compromise code quality and maintainability. It is generally advisable to use `any` sparingly and only in scenarios where the type cannot be determined or is dynamic, such as when dealing with third-party libraries or during gradual migration of legacy code.



```
let value: any = 42; // OK
value = 'hello'; // Also OK
value = {}; // Still OK
value = [];  // Still OK
value = true;  // Still OK
```

Mohammad Faisal
@mohammadfaisalkhan

Save

Unknown Data Type

It is a safer alternative to `**any**`. It represents a value whose type is not known at compile time, but unlike `any`, TypeScript still enforces type checking for `unknown`. When working with `unknown`, you must perform type checks or use type assertions to determine its type before performing operations. This prevents runtime errors and ensures type safety. The `unknown` type is particularly useful when dealing with data from external sources, user input, or any situation where the type is uncertain but you still want to ensure type correctness through explicit validation or conversion. It provides a more controlled way to handle dynamic data while maintaining TypeScript's type safety principles.

Unknown

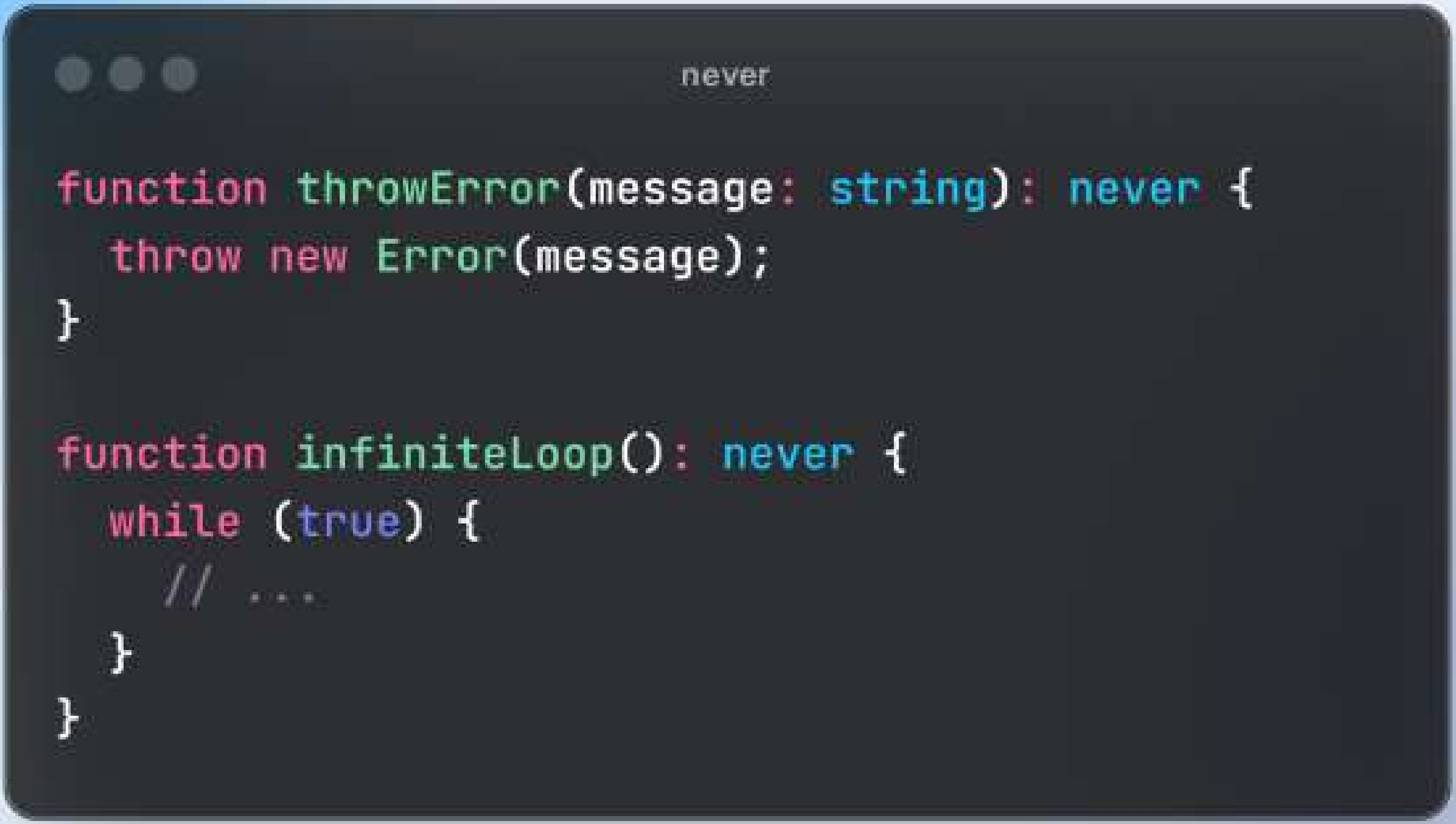
```
let value: unknown = 42; // OK
let x: number = value;
// Error: Type 'unknown' is not assignable to type 'number'

if (typeof value === 'number') {
  let x: number = value; // OK
}
```

Mohammad Faisal
@mohammadfaisalkhan

Save

Never Data Type

The `**never**` type in TypeScript represents a value that never occurs. It is primarily used as a return type for functions that always throw an exception, exit the program, or enter an infinite loop. For example, a function that always throws an error or a function with an infinite loop would have a return type of `**never**`. It is also used to represent unreachable code paths, where the control flow is not expected to ever reach certain statements. Since `never` signifies a state that can never be reached or a value that cannot exist, you cannot assign any value to a variable of type `**never**`, as it represents a condition or code path that is logically impossible.



```
function throwError(message: string): never {
  throw new Error(message);
}

function infiniteLoop(): never {
  while (true) {
    // ...
  }
}
```

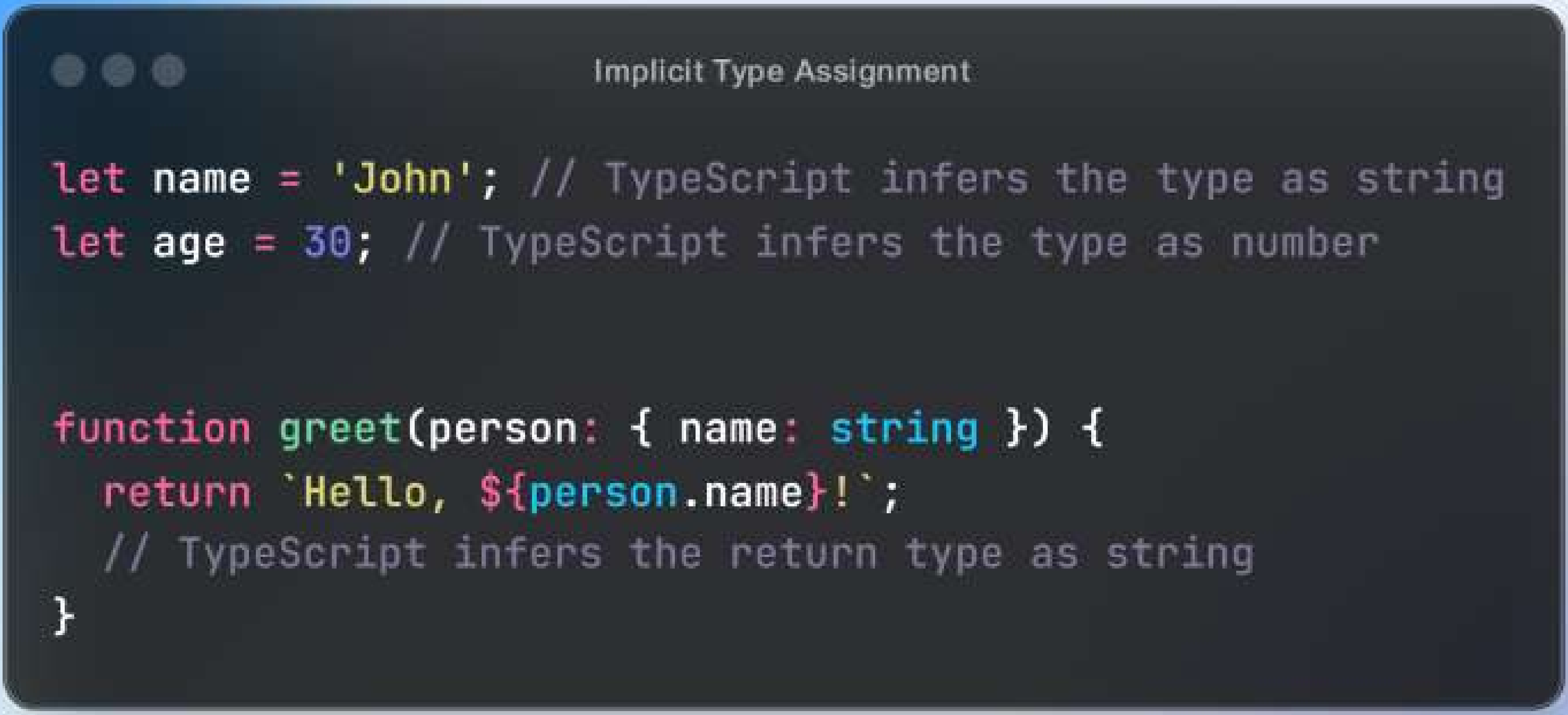
Mohammad Faisal
@mohammadfaisalkhan

Save

Type assignment

Implicit Type assignment

Implicit type assignment occurs when TypeScript infers the type of a variable, parameter, or function return value based on its value or usage. This means you don't need to explicitly declare the type; TypeScript will figure it out for you.


Implicit Type Assignment

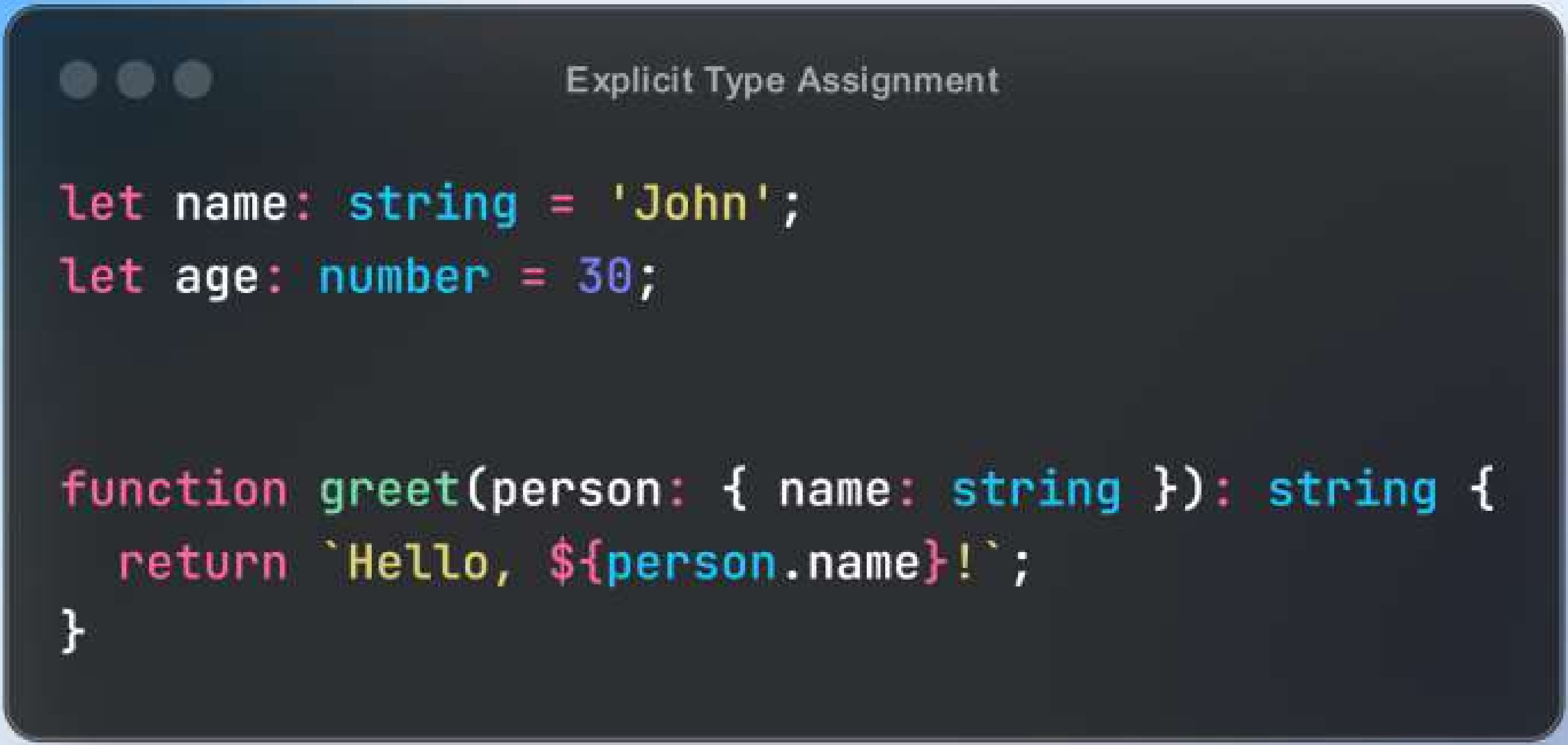
```typescript
let name = 'John'; // TypeScript infers the type as string
let age = 30; // TypeScript infers the type as number


function greet(person: { name: string }) {
  return `Hello, ${person.name}!`;
  // TypeScript infers the return type as string
}
```

Mohammad Faisal
@mohammadfaisalkhan

Save

Explicit Type assignment

Explicit type assignment involves directly specifying the type of variables, parameters, or function return values using type annotations. This approach enhances code readability and type safety by providing clear, precise type information. It is particularly useful when TypeScript's type inference is insufficient or when specific type enforcement is needed. For ex, you can declare a variable as `string` or `number`, or explicitly define a function's return type.



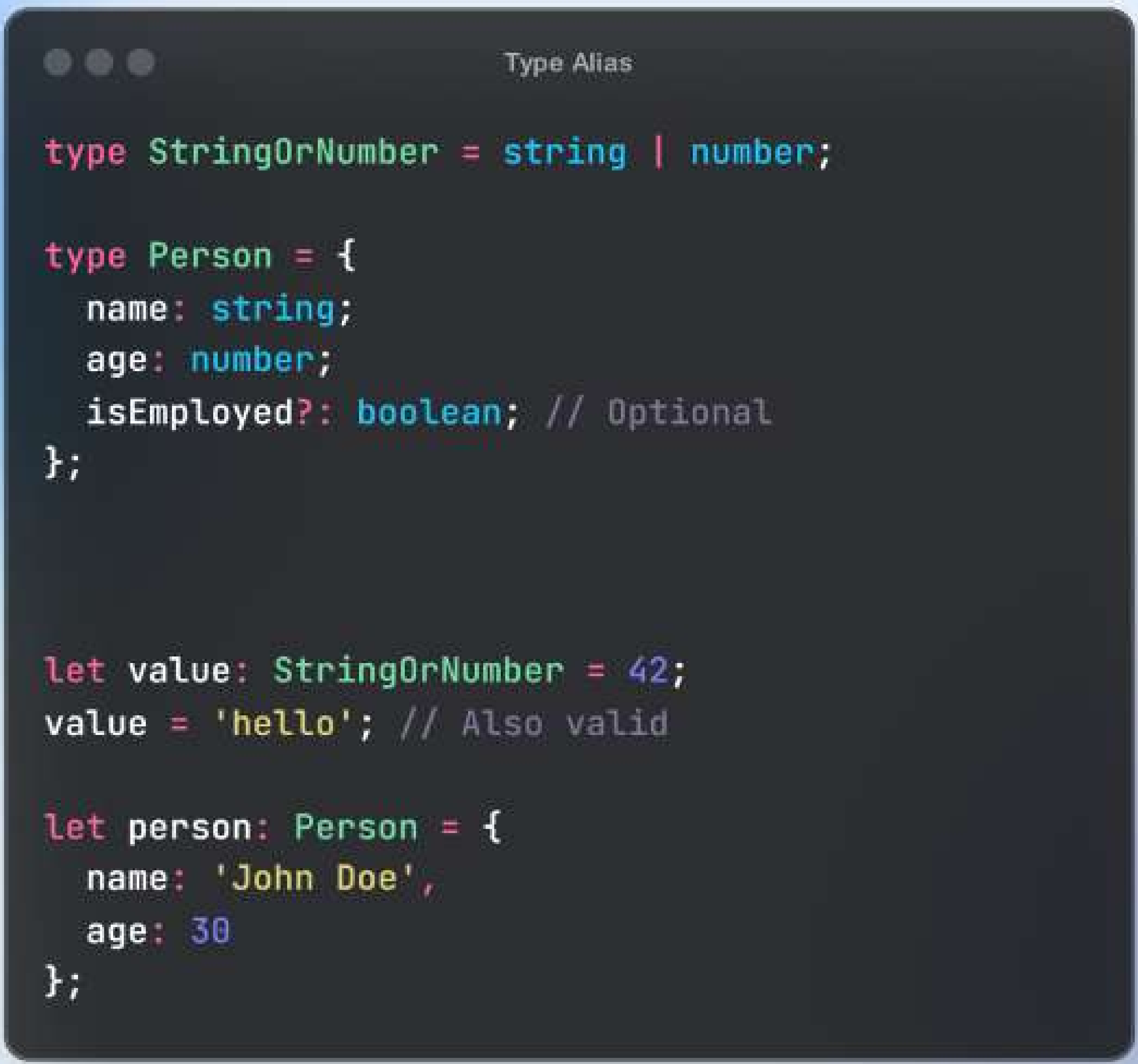
```
let name: string = 'John';
let age: number = 30;


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

Mohammad Faisal
@mohammadfaisalkhan

Save

Type Alias

Type aliases are a way to create new names for existing types. They are defined using the **type** keyword followed by the alias name and the type definition. For optional properties inside the type definition, add a **?** just after the property name before the **:** (colon)

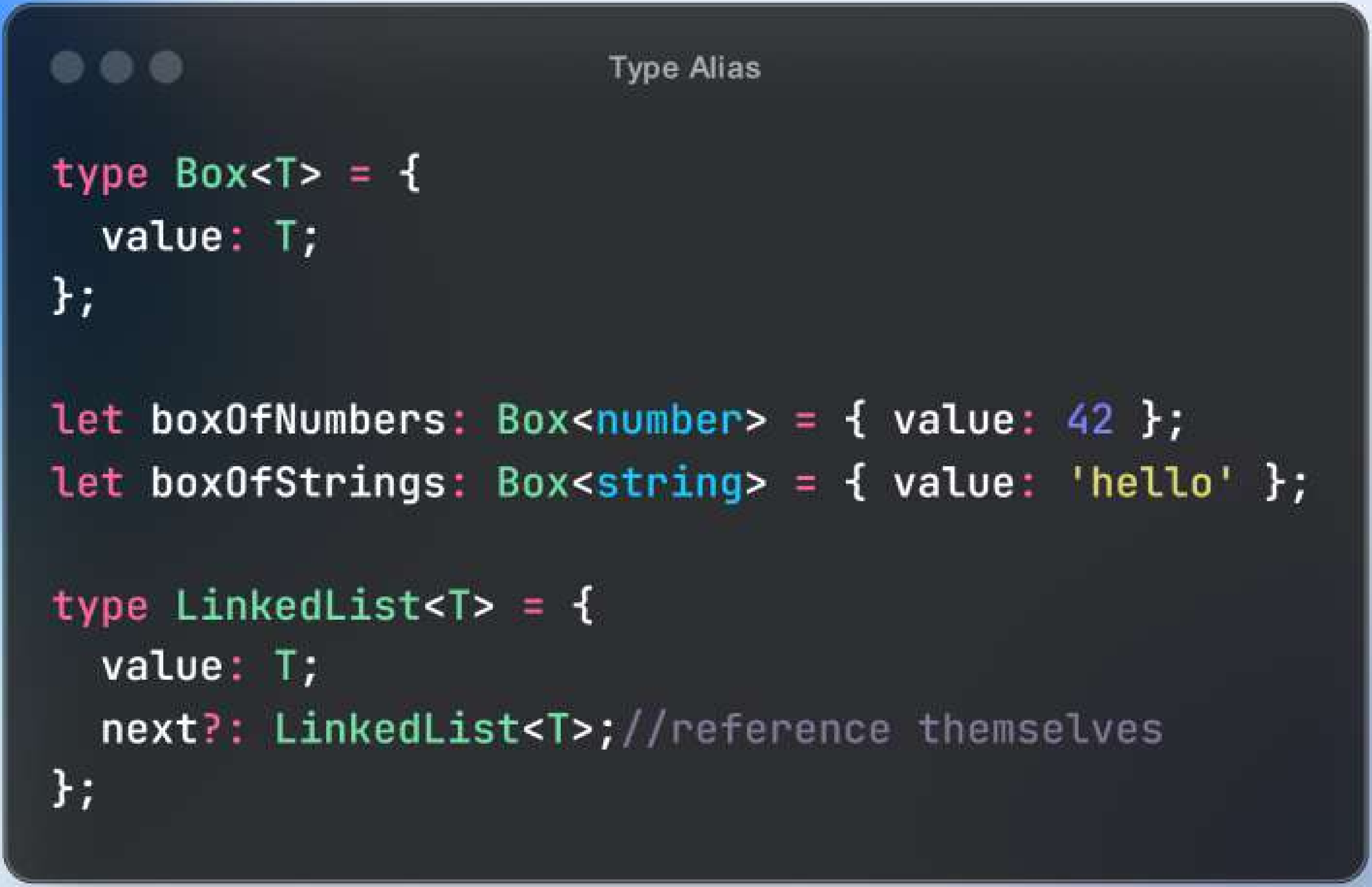


```
type StringOrNumber = string | number;

type Person = {
  name: string;
  age: number;
  isEmployed?: boolean; // Optional
};



let value: StringOrNumber = 42;
value = 'hello'; // Also valid

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

Mohammad Faisal
@mohammadfaisalkhan

Save

Type aliases can be generic also, meaning they can work with different types.
They can also reference themselves in recursive type definitions, which is useful for defining data structures like linked lists or trees.



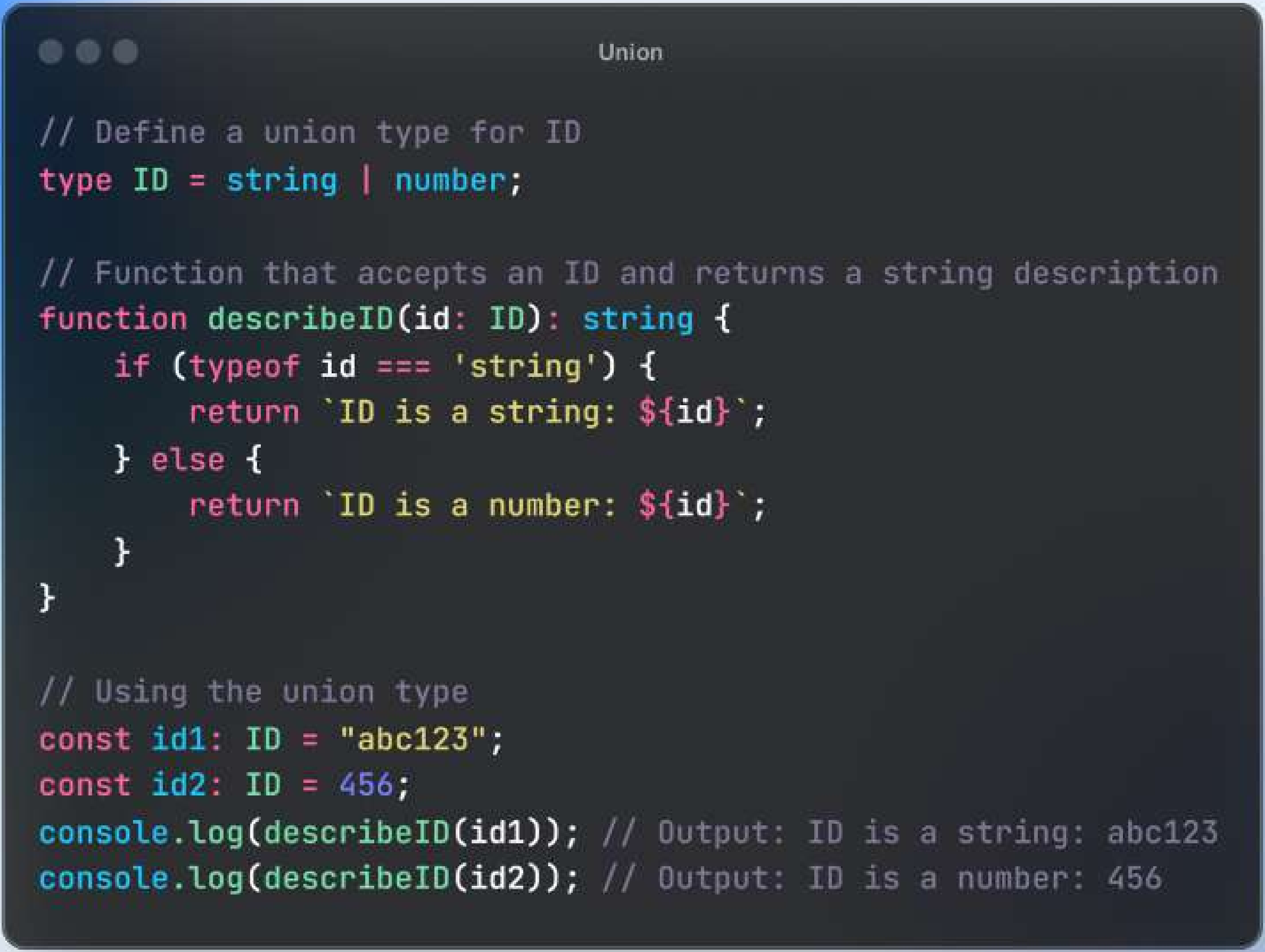
```
type Box<T> = {
  value: T;
};

let boxOfNumbers: Box<number> = { value: 42 };
let boxOfStrings: Box<string> = { value: 'hello' };

type LinkedList<T> = {
  value: T;
  next?: LinkedList<T>;//reference themselves
};
```

Mohammad Faisal
@mohammadfaisalkhan

Save

Union

A union type is a type that represents a value that can be one of several types. It is denoted by the | operator. Here, the ID type can be either a string or a number.



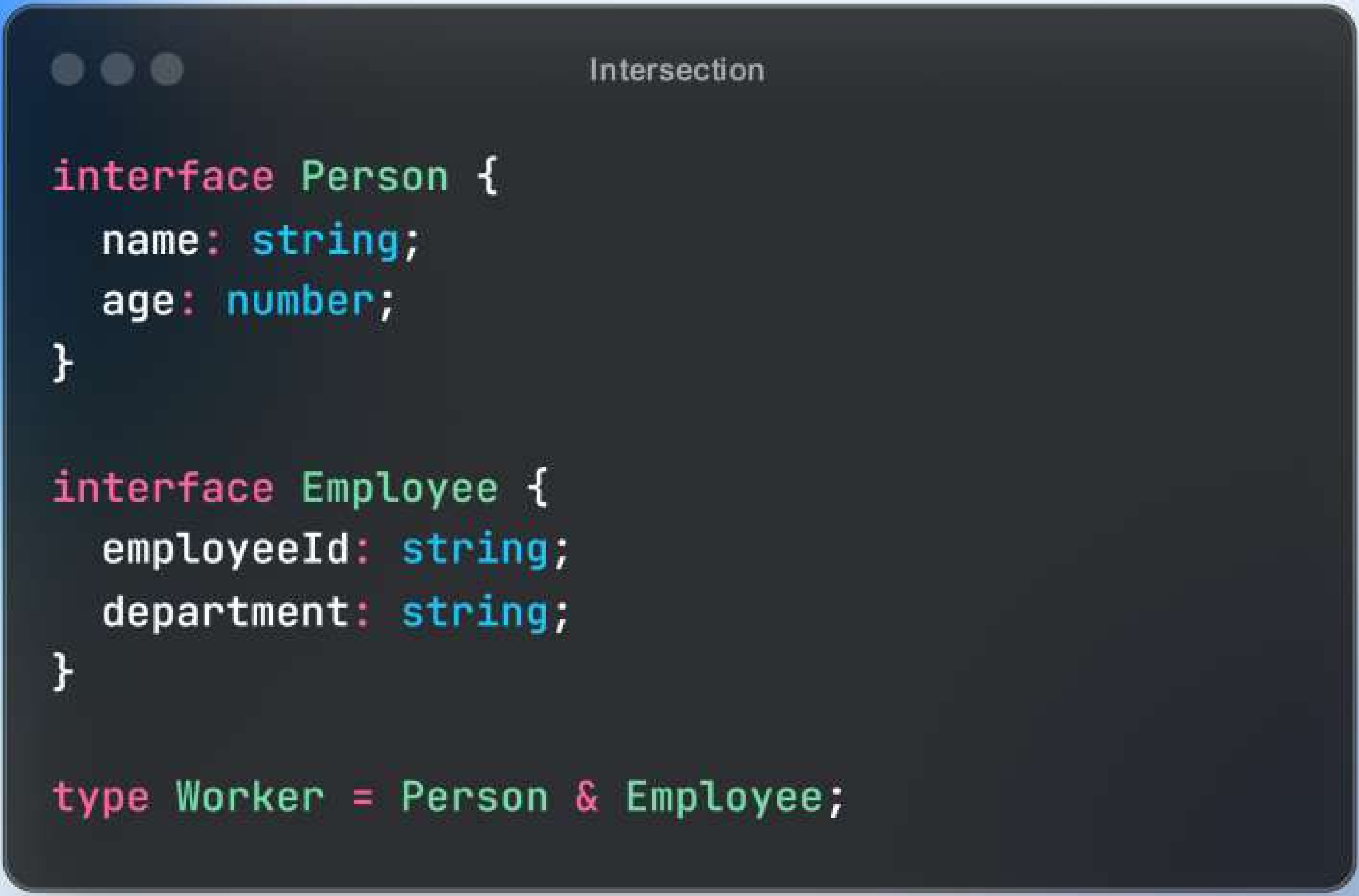
```typescript
// Define a union type for ID
type ID = string | number;

// Function that accepts an ID and returns a string description
function describeID(id: ID): string {
    if (typeof id === 'string') {
        return `ID is a string: ${id}`;
    } else {
        return `ID is a number: ${id}`;
    }
}

// Using the union type
const id1: ID = "abc123";
const id2: ID = 456;
console.log(describeID(id1)); // Output: ID is a string: abc123
console.log(describeID(id2)); // Output: ID is a number: 456
```

Mohammad Faisal
@mohammadfaisalkhan

Save

Intersection

An intersection type is a type that combines multiple types into one. It is denoted by the & operator. In this case, the Worker type has all the properties of both Person and Employee types.



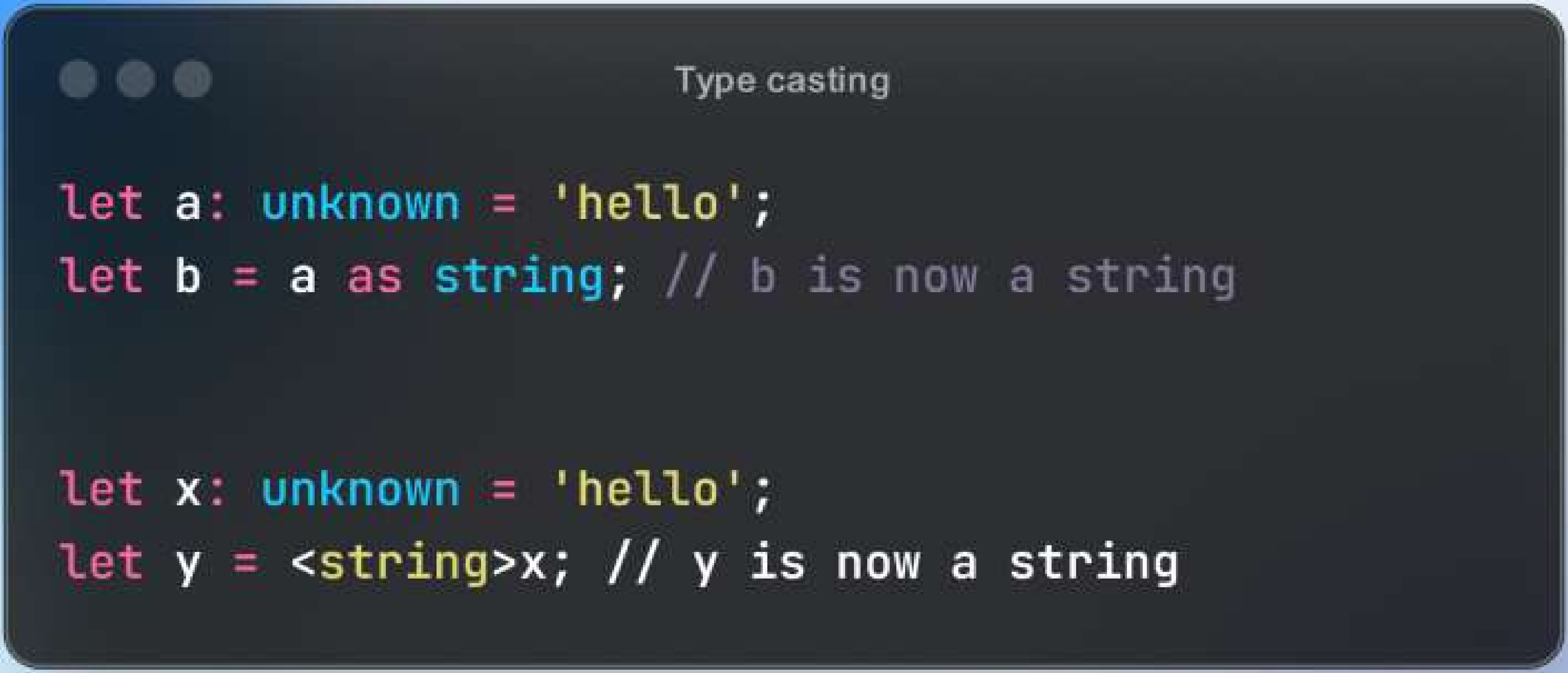
```typescript
interface Person {
  name: string;
  age: number;
}

interface Employee {
  employeeId: string;
  department: string;
}

type Worker = Person & Employee;
```

Mohammad Faisal
@mohammadfaisalkhan

Save

Type casting

Type casting in TypeScript is a way to override the type inference system and explicitly specify the type of a value.
TypeScript has two types of type casting: the **as** syntax and the angle-bracket **<>** syntax.



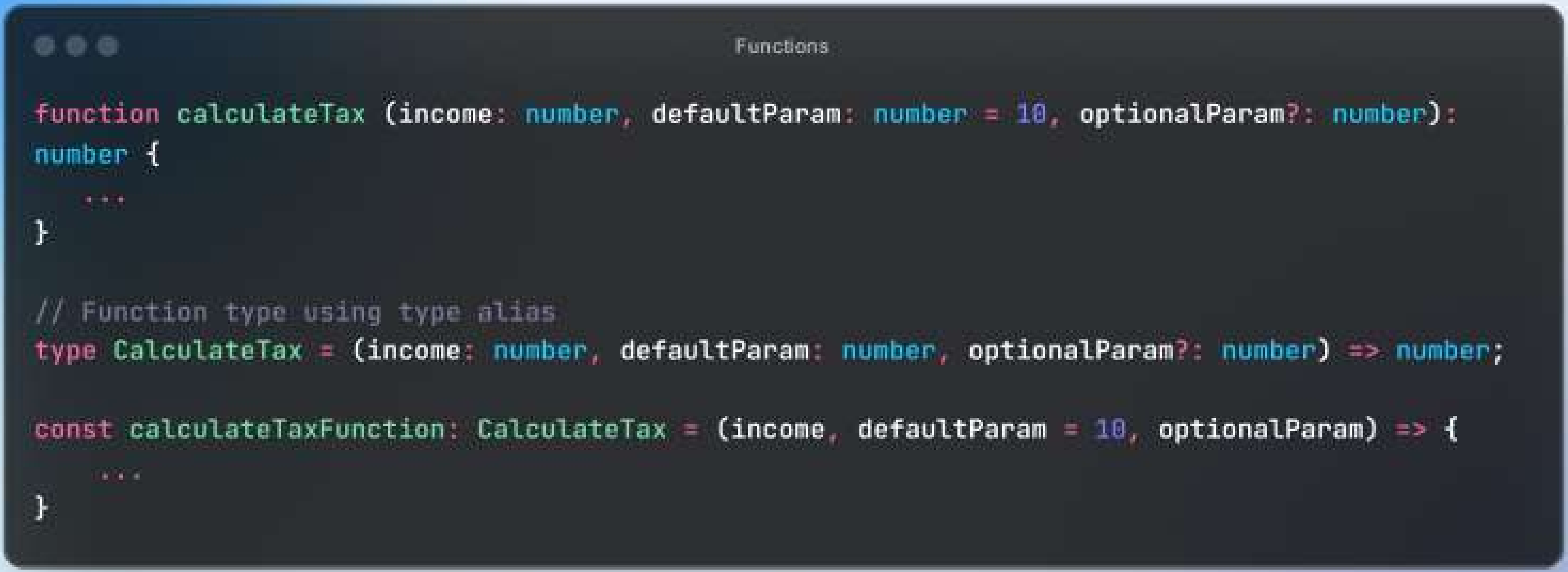
```
let a: unknown = 'hello';
let b = a as string; // b is now a string


let x: unknown = 'hello';
let y = <string>x; // y is now a string
```

Mohammad Faisal
@mohammadfaisalkhan

Save

Functions

In TypeScript, you specify function parameter types and return types using a similar syntax to variable declarations. For parameters with default values, the default is placed after the type annotation, while optional parameters use **?** before the type. The return type is declared after the function's parameter list. You can give return type **void** if your function doesn't return any value..



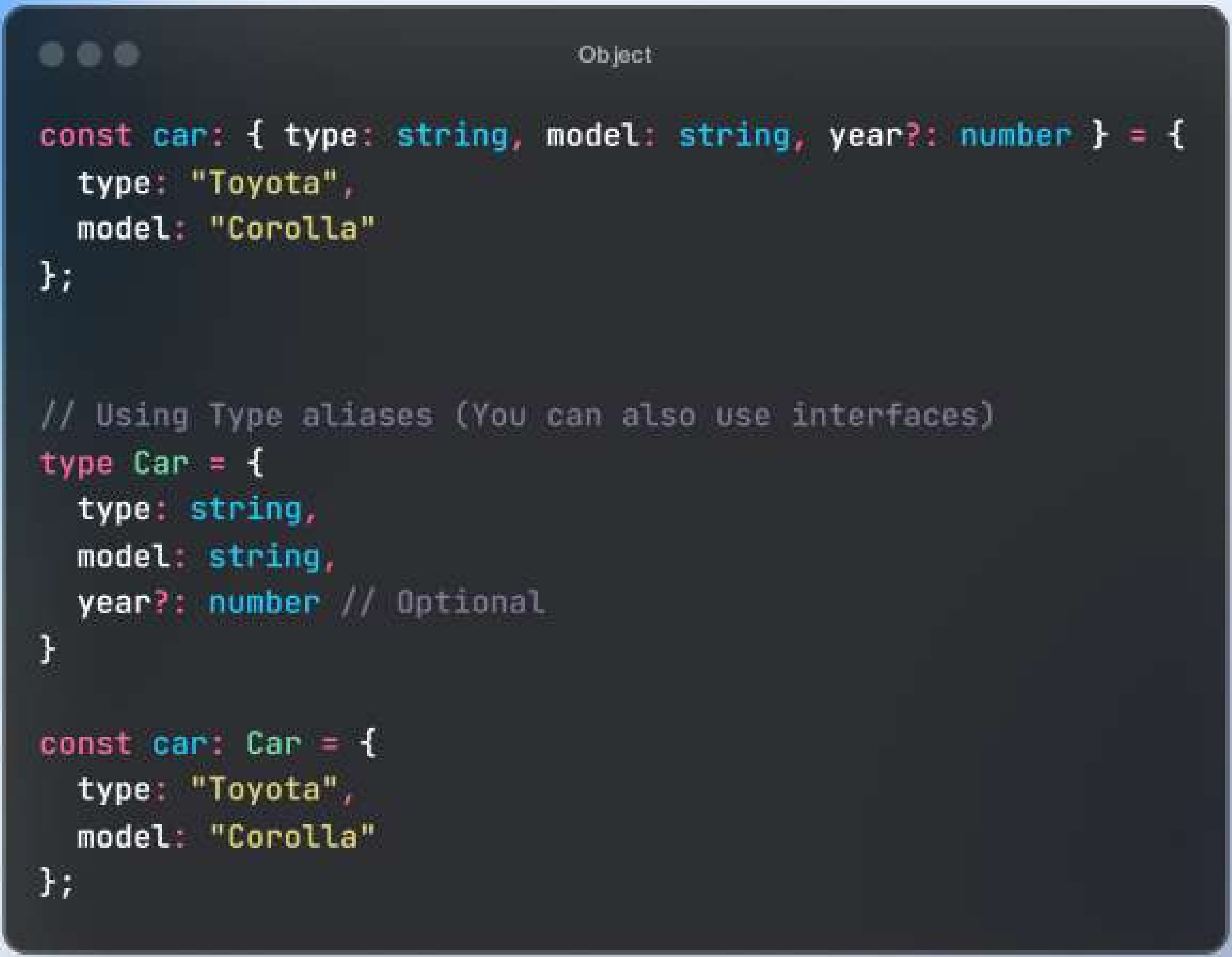
```
function calculateTax (income: number, defaultParam: number = 10, optionalParam?: number):
number {
    ...
}

// Function type using type alias
type CalculateTax = (income: number, defaultParam: number, optionalParam?: number) => number;

const calculateTaxFunction: CalculateTax = (income, defaultParam = 10, optionalParam) => {
    ...
}
```

Mohammad Faisal
@mohammadfaisalkhan

Save

Objects

TypeScript has a specific syntax for typing objects. Specify the types of each object properties in the curly braces after a **:** after the object name. Optional properties can be marked with a **?** just before the **:** and type. You can write object types separately, and they can even be reused, using type aliases or interfaces.



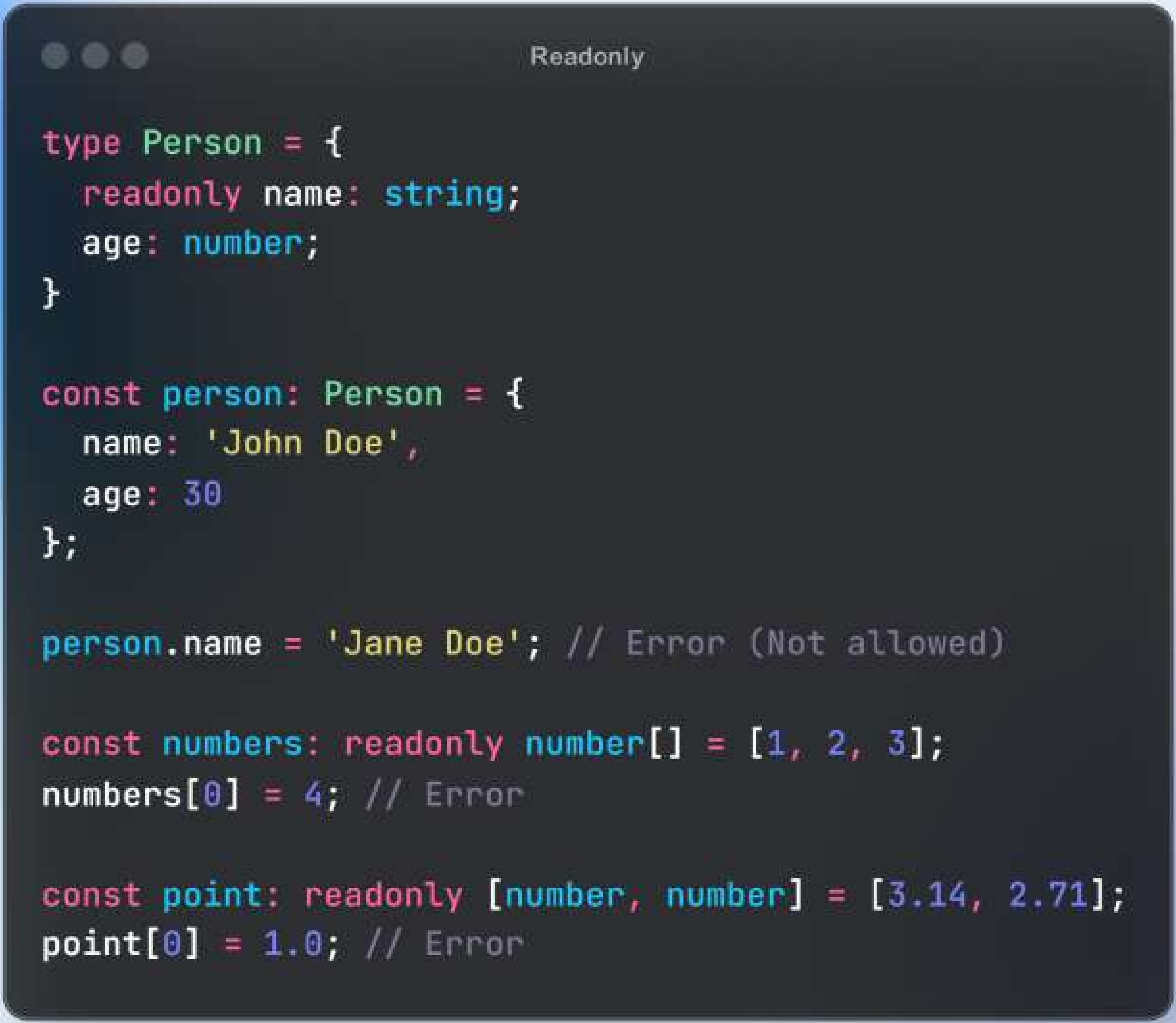
```typescript
const car: { type: string, model: string, year?: number } = {
  type: "Toyota",
  model: "Corolla"
};


// Using Type aliases (You can also use interfaces)
type Car = {
  type: string,
  model: string,
  year?: number // Optional
}

const car: Car = {
  type: "Toyota",
  model: "Corolla"
};
```

Mohammad Faisal
@mohammadfaisalkhan

Save

Readonly

The **readonly** keyword is used to mark properties or indexes of an object or array as read-only. This means that the values of these properties or indexes cannot be reassigned after the object or array is created.



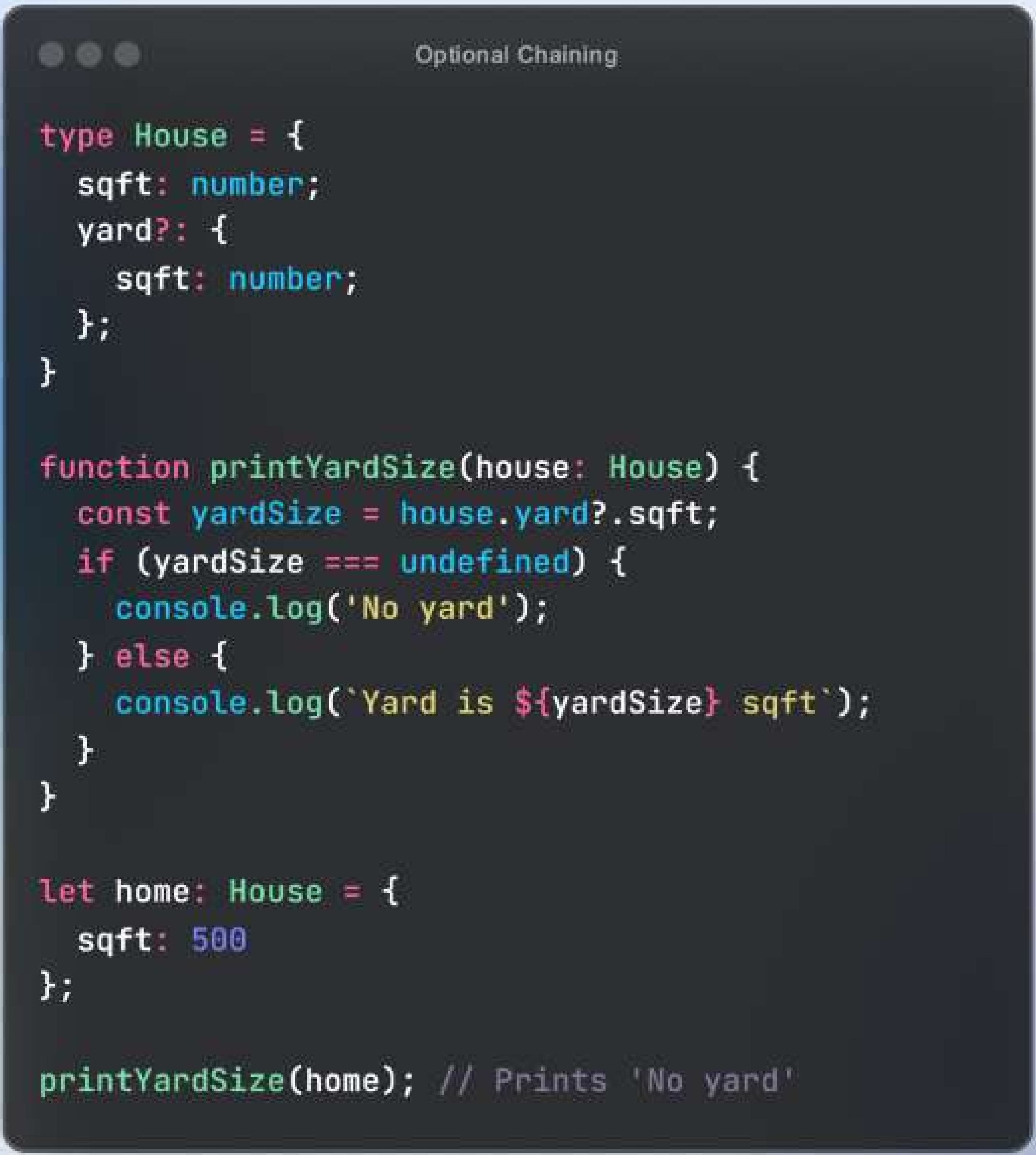
```typescript
type Person = {
  readonly name: string;
  age: number;
}

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

person.name = 'Jane Doe'; // Error (Not allowed)

const numbers: readonly number[] = [1, 2, 3];
numbers[0] = 4; // Error

const point: readonly [number, number] = [3.14, 2.71];
point[0] = 1.0; // Error
```

Mohammad Faisal
@mohammadfaisalkhan

Save

Optional Chaining

It allows accessing properties on an object, that may or may not exist, with a compact syntax. It can be used with the **?.** operator when accessing properties.



```
type House = {
  sqft: number;
  yard?: {
    sqft: number;
  };
}

function printYardSize(house: House) {
  const yardSize = house.yard?.sqft;
  if (yardSize === undefined) {
    console.log('No yard');
  } else {
    console.log(`Yard is ${yardSize} sqft`);
  }
}

let home: House = {
  sqft: 500
};

printYardSize(home); // Prints 'No yard'
```

Mohammad Faisal
@mohammadfaisalkhan

Save

Nullish Coalescence

Nullish Coalescence provides a concise way to handle null or undefined values. It is a shorthand for the common pattern of using the logical OR operator **(||)** to provide a default value when dealing with nullable types. This is useful when other falsy values can occur in the expression but are still valid. It can be used with the **??** operator in an expression.

Nullish Coalescence

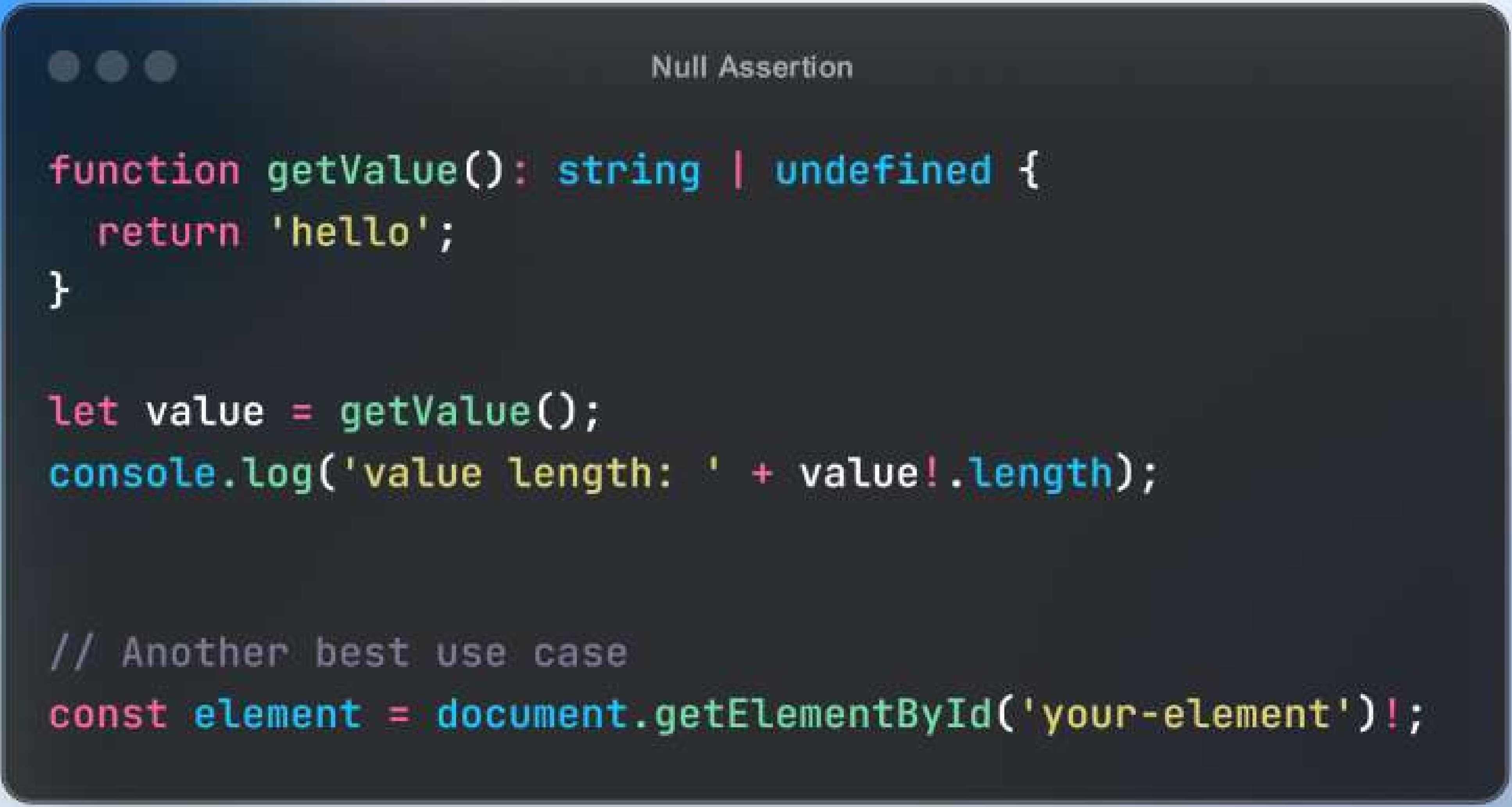
```
function printMileage(mileage: number | null | undefined) {
  console.log(`Mileage: ${mileage ?? 'Not Available'}`);
}

printMileage(null); // Prints 'Mileage: Not Available'
printMileage(0); // Prints 'Mileage: 0'
```

Mohammad Faisal
@mohammadfaisalkhan

Save

Null Assertion

TypeScript's inference system isn't perfect, there are times when it makes sense to ignore a value's possibility of being **null** or **undefined**. Hence, it has a non-null assertion that you can use when you are sure that the value is never null by adding the **!** operator to the end of your statement.



Null Assertion

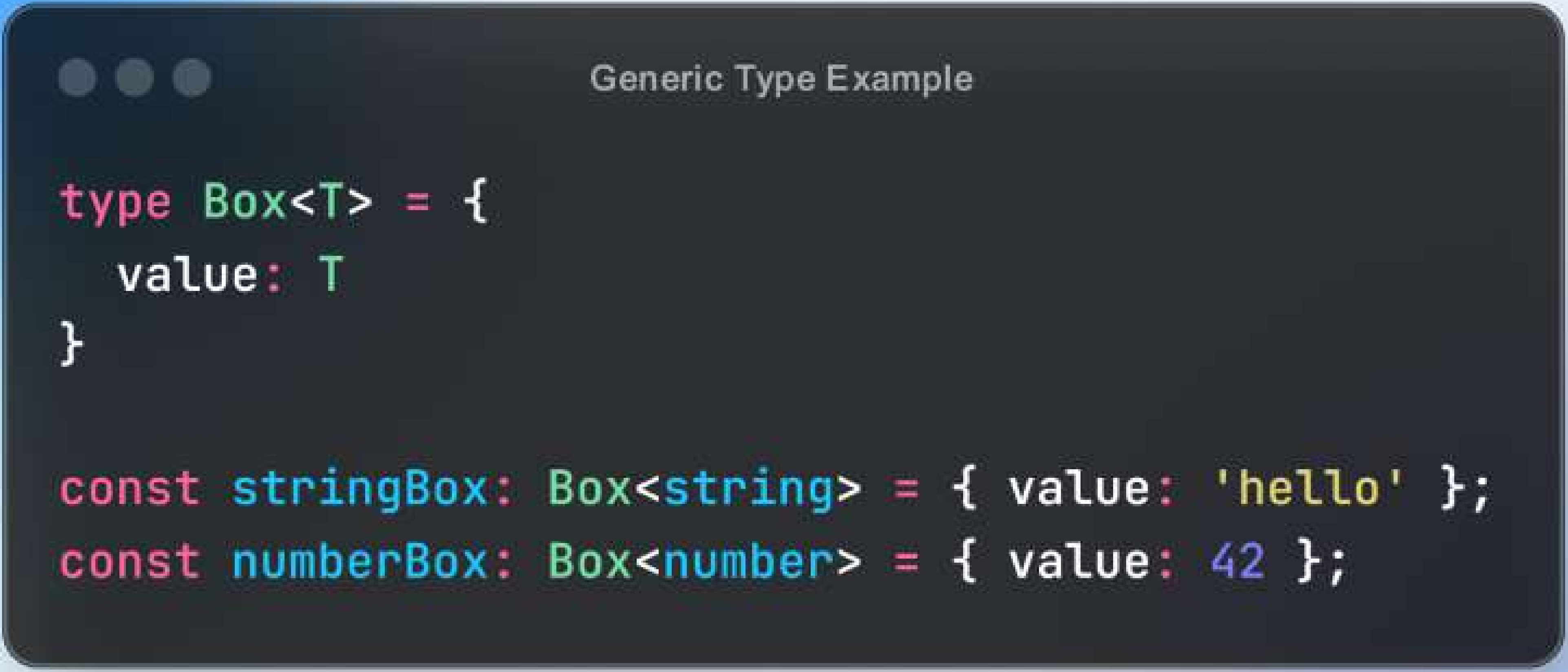
```
function getValue(): string | undefined {
  return 'hello';
}

let value = getValue();
console.log('value length: ' + value!.length);


// Another best use case
const element = document.getElementById('your-element')!;
```

Mohammad Faisal
@mohammadfaisalkhan

Save

Generics

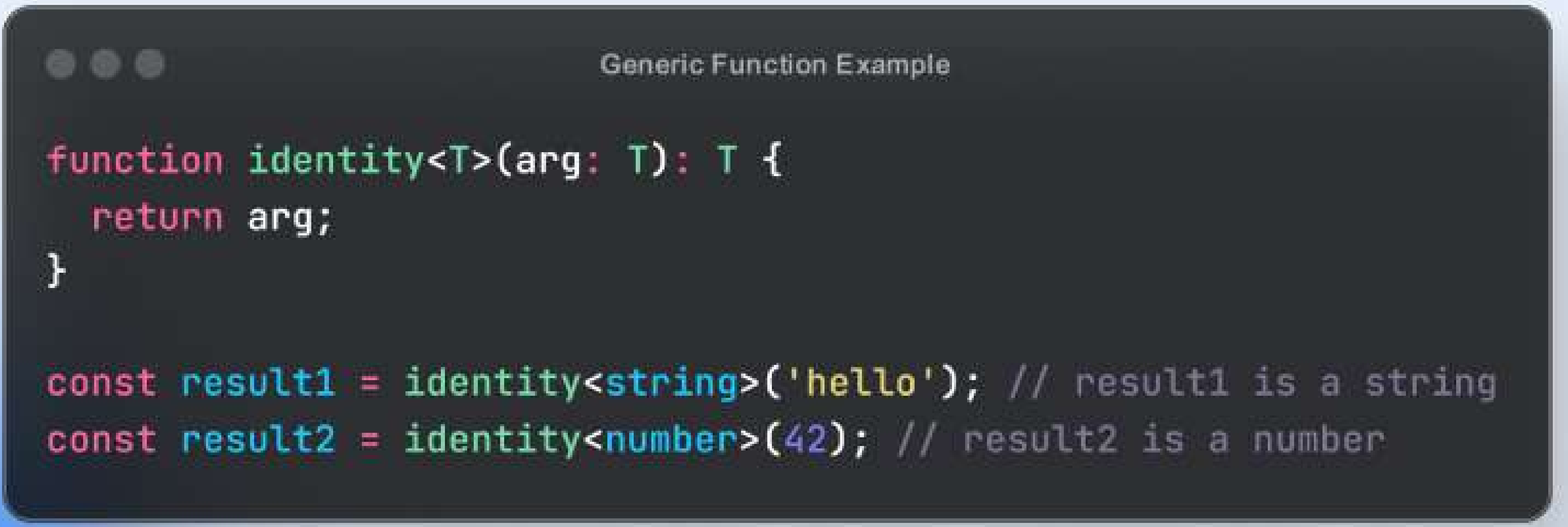
Generics in TypeScript allow you to create reusable components that can work with a variety of data types instead of a single data type. They provide a way to define type variables that can be used to create classes, interfaces, functions, and other types that are parameterized by one or more types.



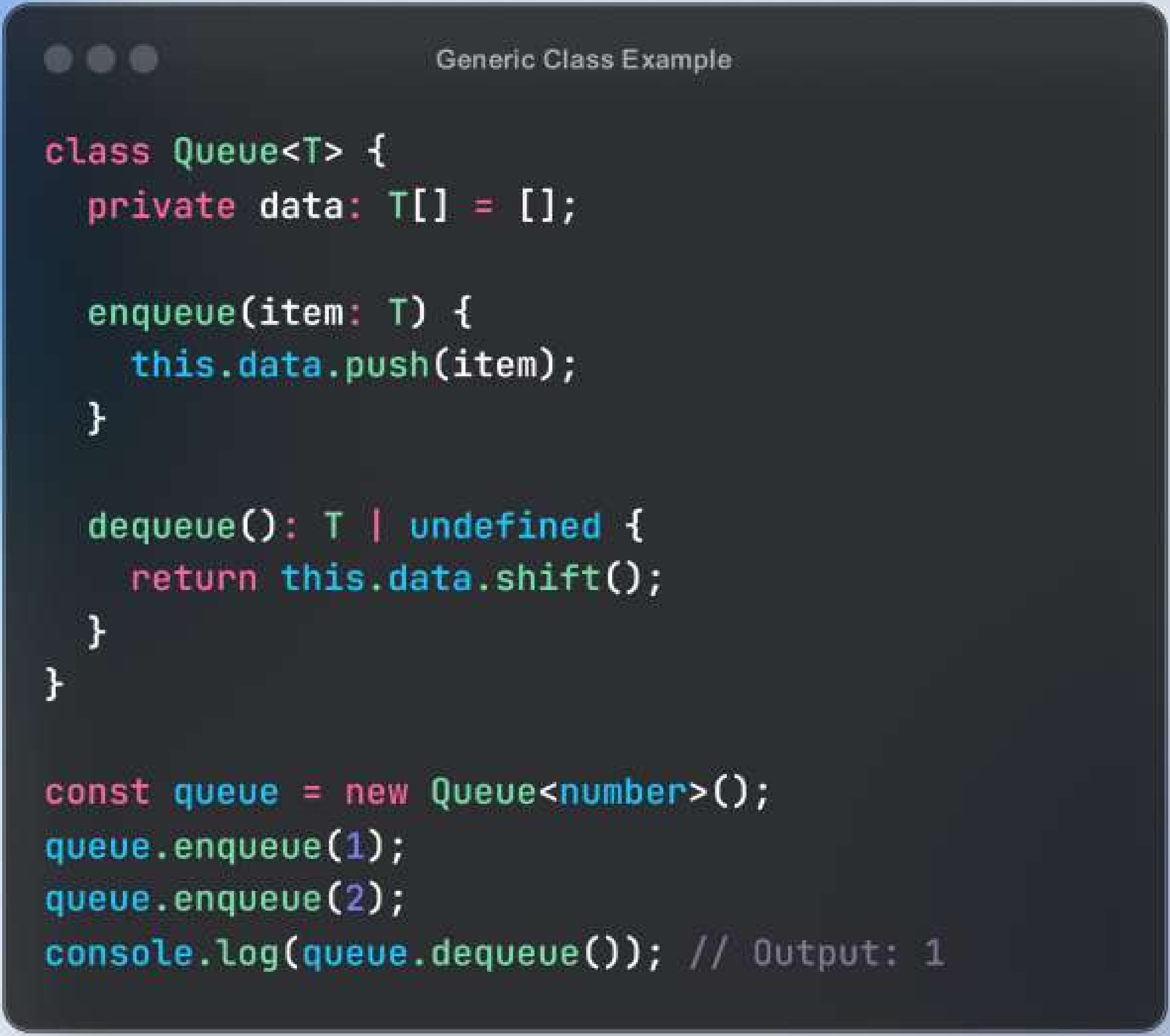
```typescript
type Box<T> = {
  value: T
}

const stringBox: Box<string> = { value: 'hello' };
const numberBox: Box<number> = { value: 42 };
```

Mohammad Faisal
@mohammadfaisalkhan

Save


Generic Function Example

```
function identity<T>(arg: T): T {
  return arg;
}

const result1 = identity<string>('hello'); // result1 is a string
const result2 = identity<number>(42); // result2 is a number
```


Generic Class Example

```
class Queue<T> {
  private data: T[] = [];

  enqueue(item: T) {
    this.data.push(item);
  }

  dequeue(): T | undefined {
    return this.data.shift();
  }
}

const queue = new Queue<number>();
queue.enqueue(1);
queue.enqueue(2);
console.log(queue.dequeue()); // Output: 1
```

Mohammad Faisal
@mohammadfaisalkhan

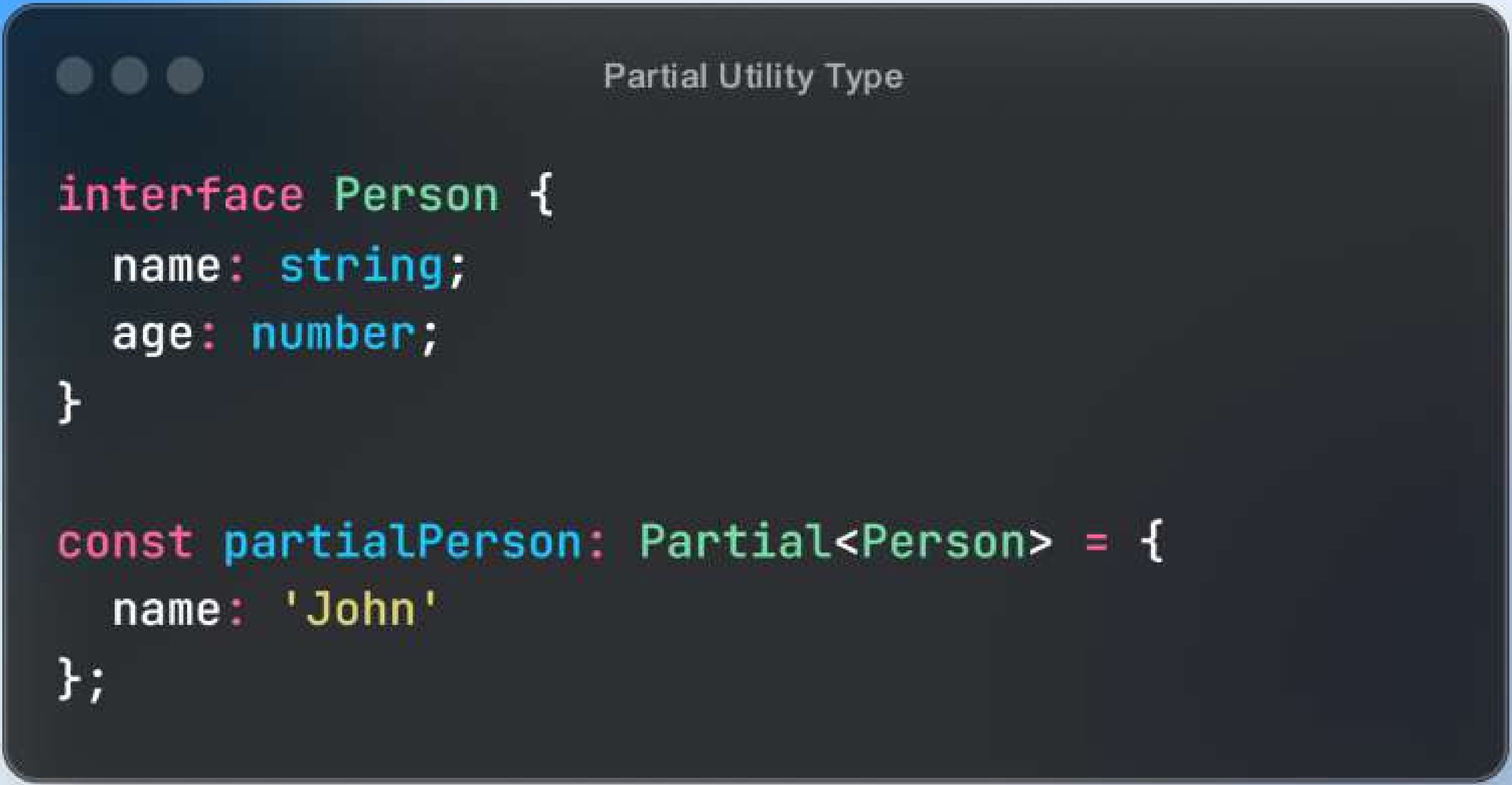
Save

Utility Types

They can be used to manipulate and transform existing types. Below are the most used one

Partial Utility Type

It transforms all properties of a given type into optional properties. This can be useful where you want to create objects that are a subset of a larger type or to handle configurations and updates flexibly.



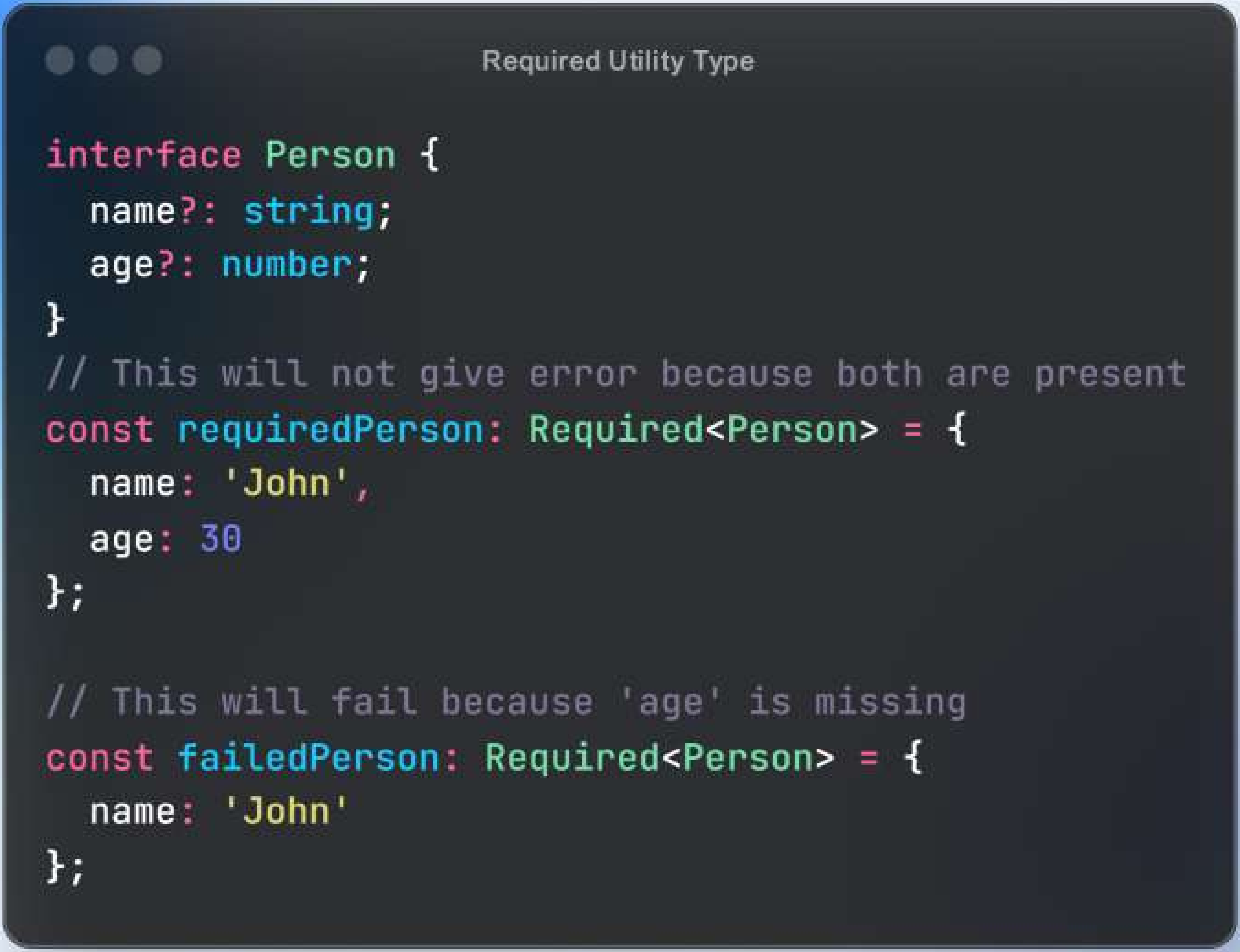
```
interface Person {
  name: string;
  age: number;
}

const partialPerson: Partial<Person> = {
  name: 'John'
};
```

Mohammad Faisal
@mohammadfaisalkhan

Save

Required Utility Type

It transforms all optional properties in a given type into required properties. This is particularly useful when you need to ensure that certain properties are explicitly provided and not left undefined, which can help enforce stricter type safety in your code.



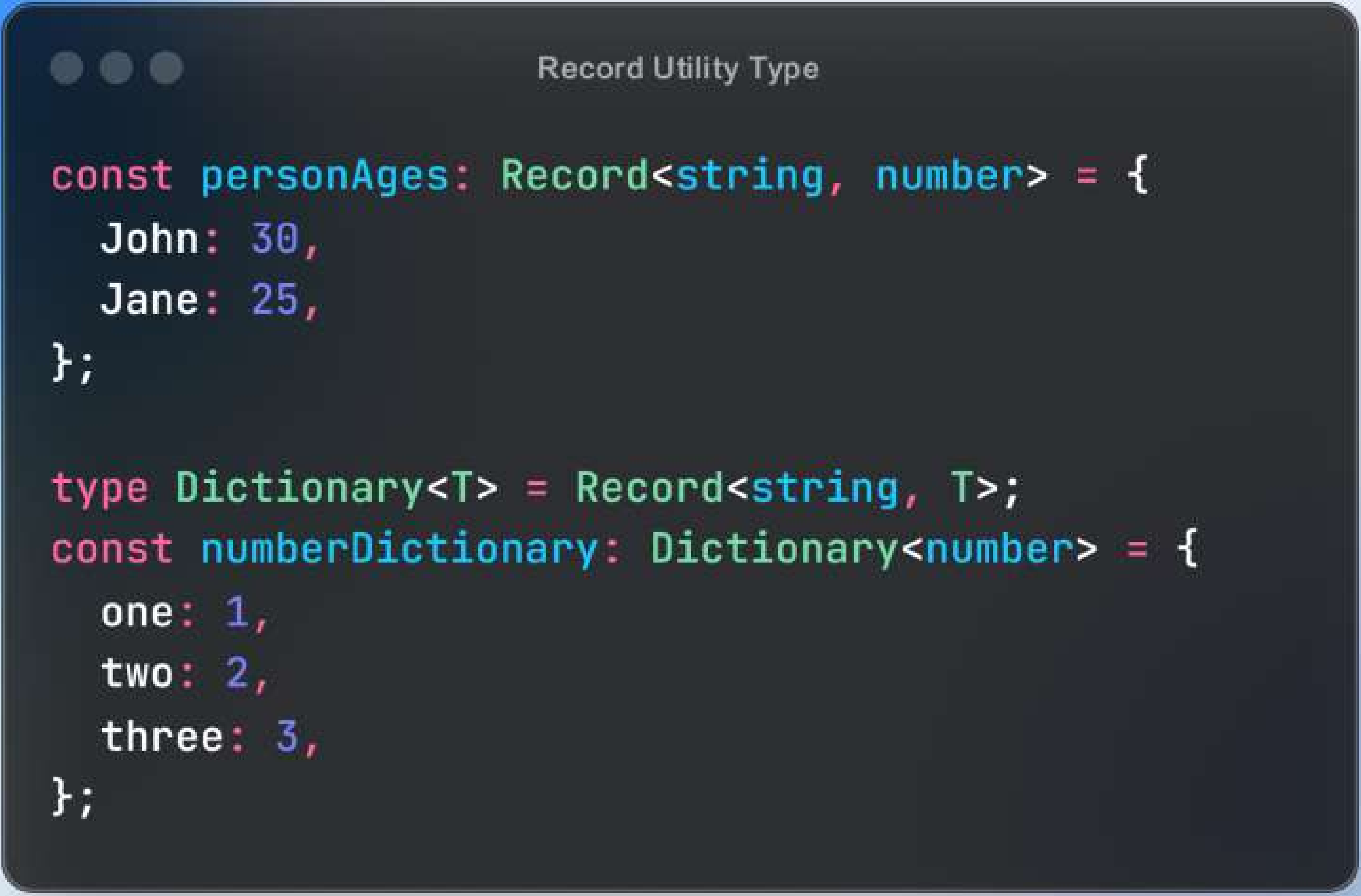
```typescript
interface Person {
  name?: string;
  age?: number;
}
// This will not give error because both are present
const requiredPerson: Required<Person> = {
  name: 'John',
  age: 30
};

// This will fail because 'age' is missing
const failedPerson: Required<Person> = {
  name: 'John'
};
```

Mohammad Faisal
@mohammadfaisalkhan

Save

Record Utility Type

The **Record** utility type is a TypeScript built-in type that creates an object type with specified keys and values. It's especially useful when you need to create objects where the keys are of a certain type, and the values are of another type.


Record Utility Type

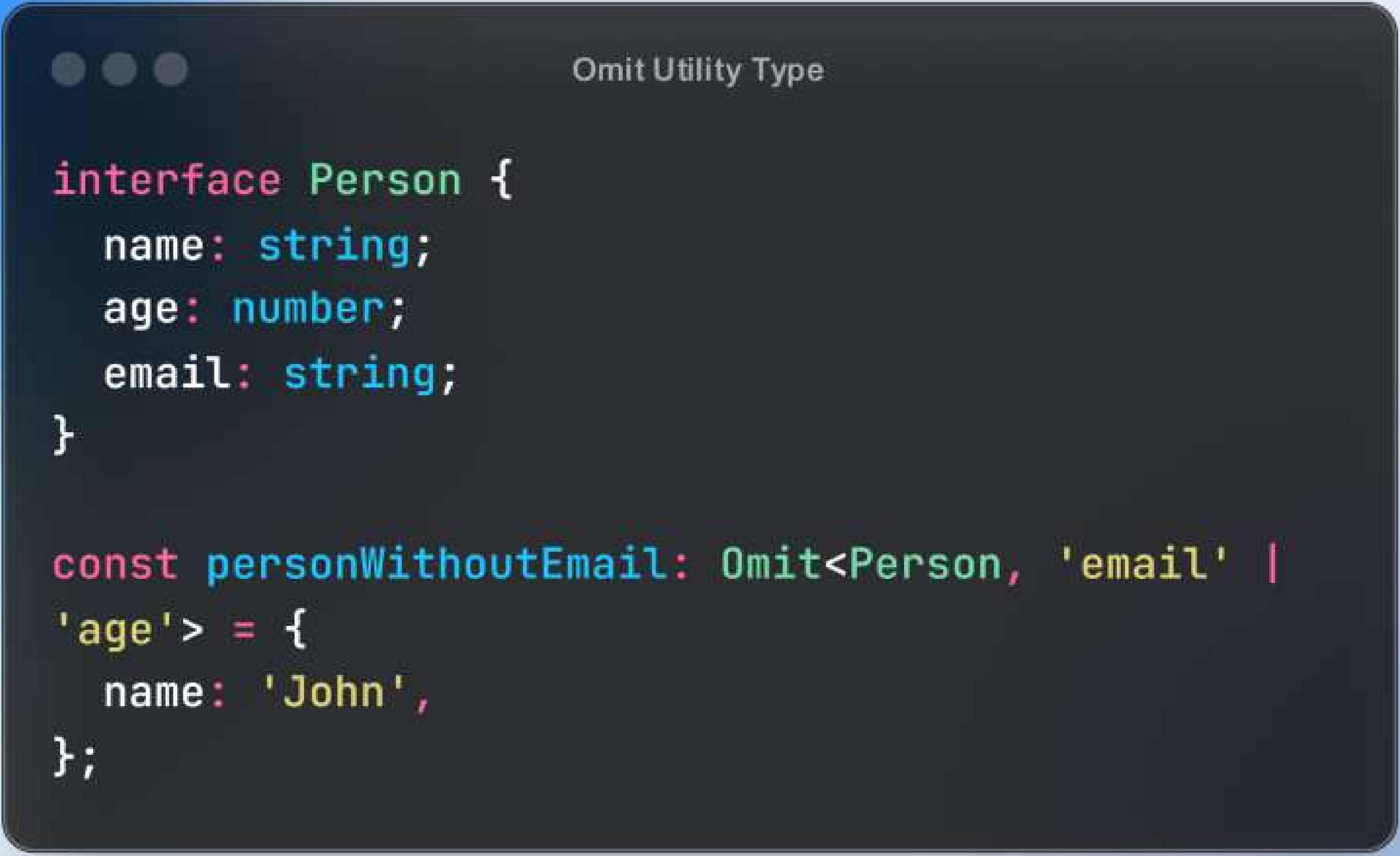
```typescript
const personAges: Record<string, number> = {
  John: 30,
  Jane: 25,
};

type Dictionary<T> = Record<string, T>;
const numberDictionary: Dictionary<number> = {
  one: 1,
  two: 2,
  three: 3,
};
```

Mohammad Faisal
@mohammadfaisalkhan

Save

Omit Utility Type

The **Omit** utility type is used to construct a type by removing one or more properties from another type. This is handy when you want to create a variant of a type without certain fields, such as for forms, API responses, or filtered data.



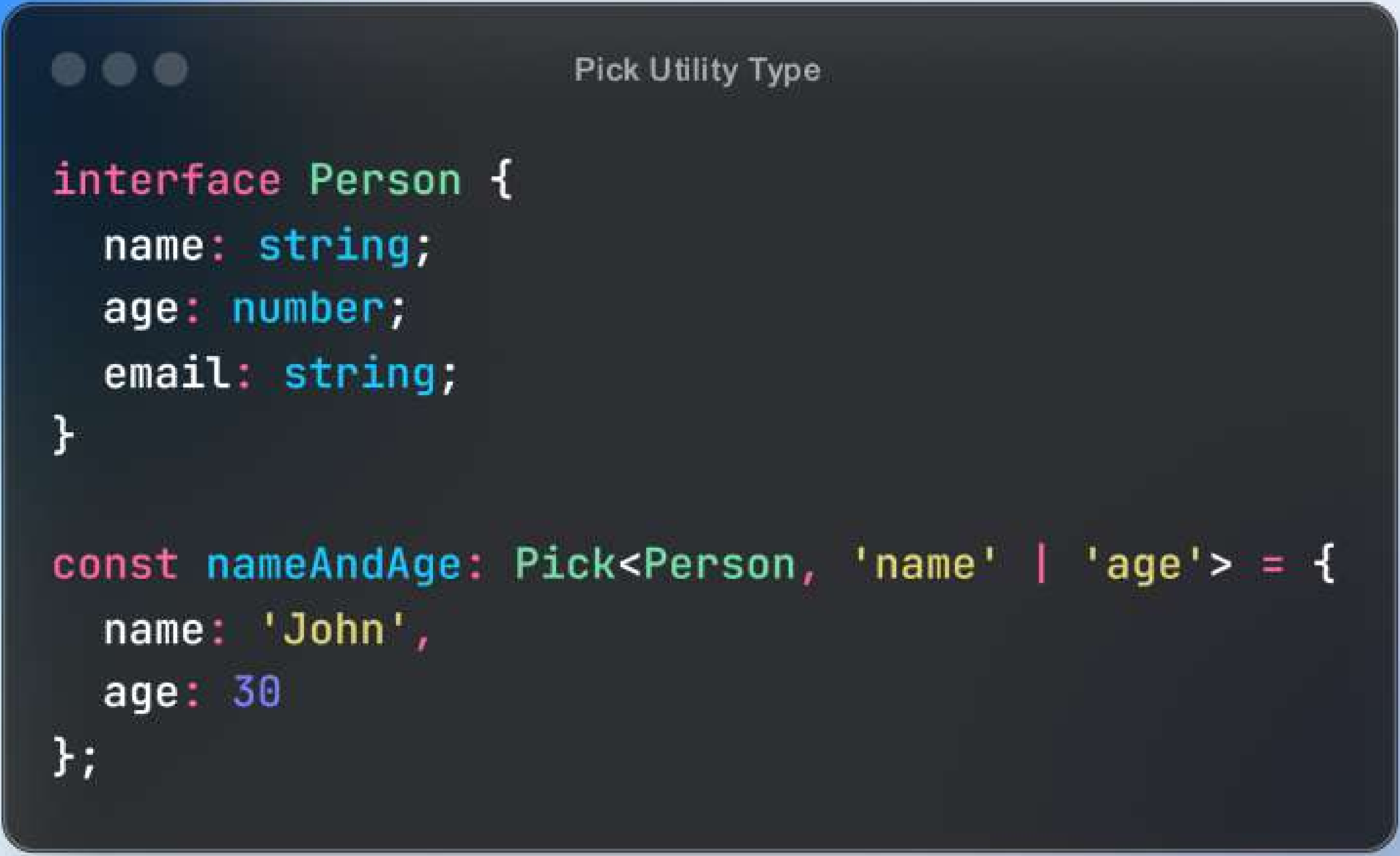
```
interface Person {
  name: string;
  age: number;
  email: string;
}

const personWithoutEmail: Omit<Person, 'email' |
'age'> = {
  name: 'John',
};
```

Mohammad Faisal
@mohammadfaisalkhan

Save

Pick Utility Type

The **Pick** utility type constructs a new type by choosing a subset of properties from an existing type. This is useful for scenarios where you need to work with only specific properties of a type without affecting the original type's structure.



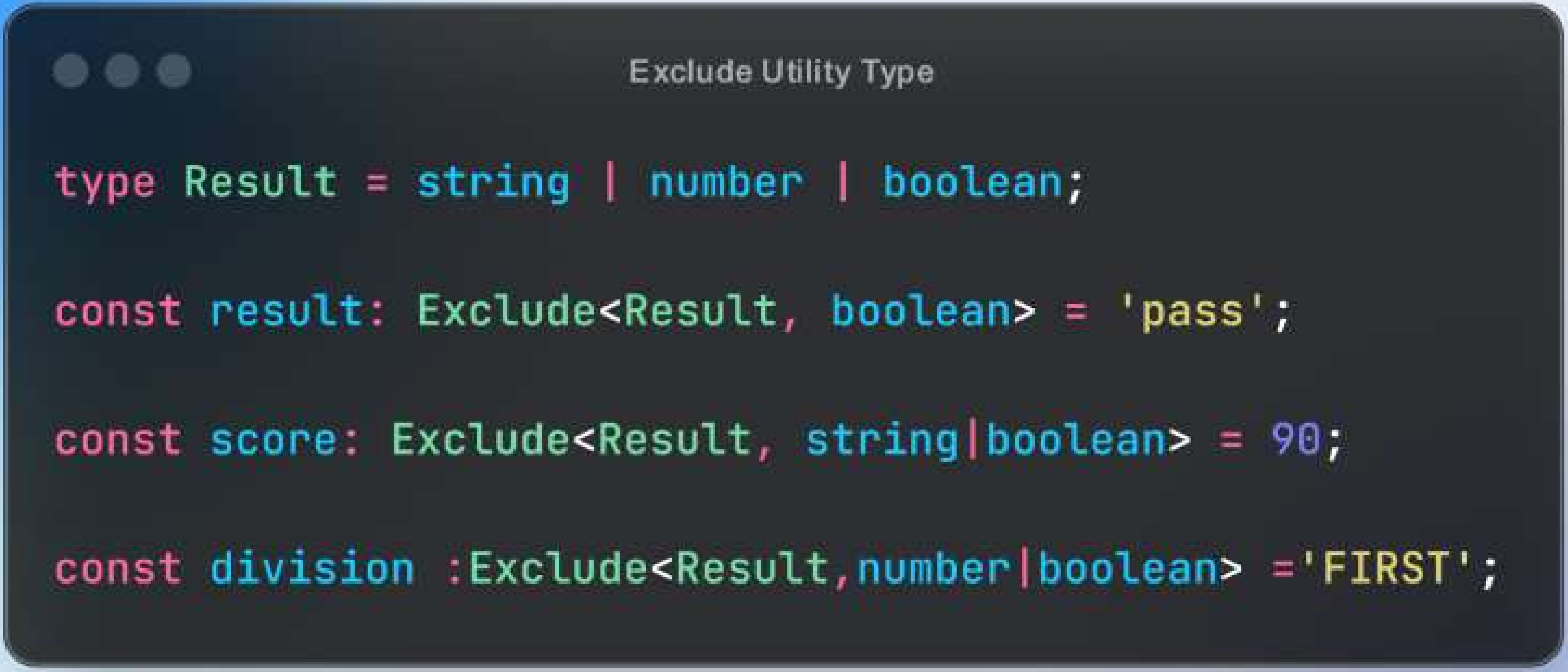
```typescript
interface Person {
  name: string;
  age: number;
  email: string;
}

const nameAndAge: Pick<Person, 'name' | 'age'> = {
  name: 'John',
  age: 30
};
```

Mohammad Faisal
@mohammadfaisalkhan

Save

Exclude Utility Type

The **Exclude** utility type constructs a new type by excluding specific types from a union type. It’s useful for scenarios where you need to refine a union type by removing certain members, thus creating a more focused or restricted type.



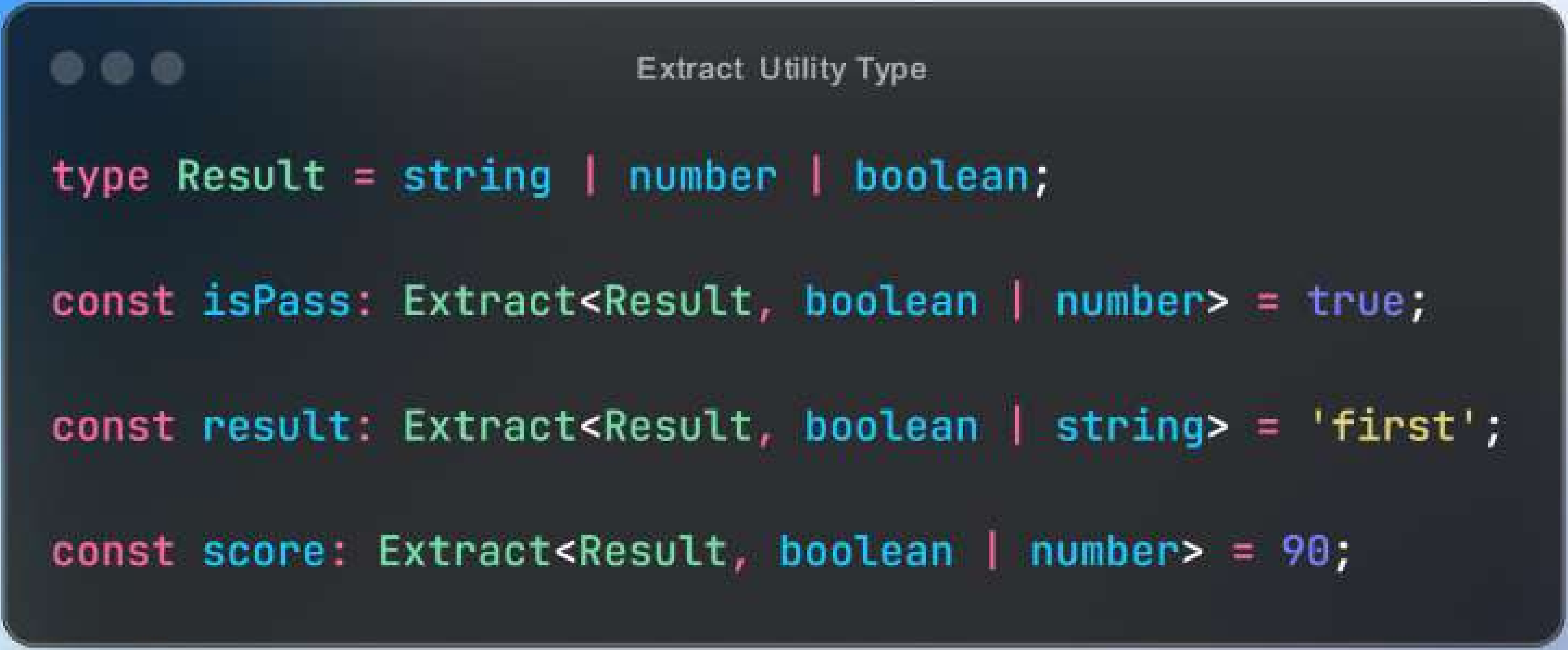
```
type Result = string | number | boolean;

const result: Exclude<Result, boolean> = 'pass';

const score: Exclude<Result, string|boolean> = 90;

const division :Exclude<Result,number|boolean> ='FIRST';
```

Mohammad Faisal
@mohammadfaisalkhan

Save

Extract Utility Type

The **Extract** utility type constructs a new type by selecting types from a union that are assignable to a specified type. It's useful when you want to filter or extract a portion of a union type based on specific criteria, thereby creating a new, more focused type.



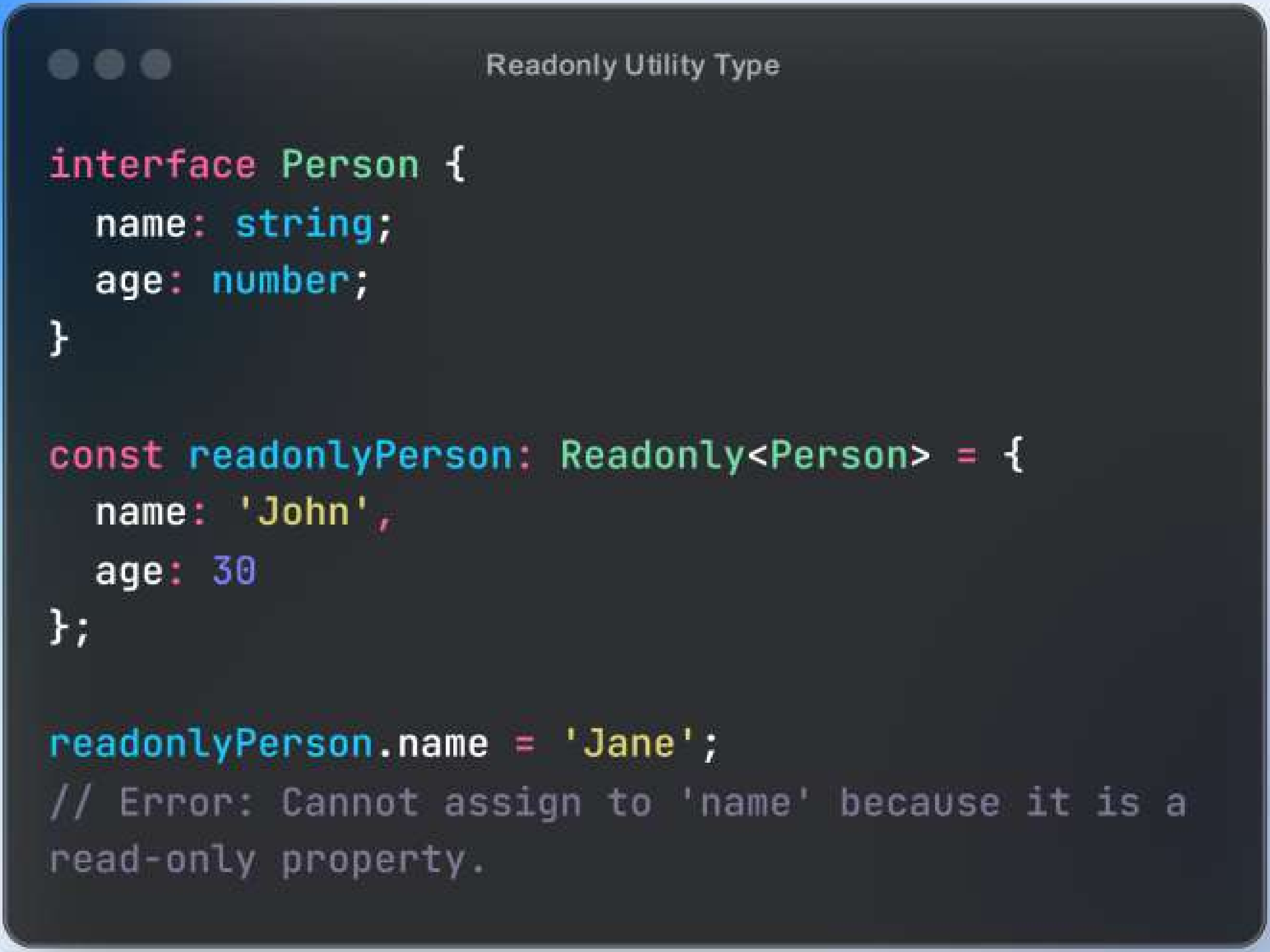
```typescript
type Result = string | number | boolean;

const isPass: Extract<Result, boolean | number> = true;

const result: Extract<Result, boolean | string> = 'first';

const score: Extract<Result, boolean | number> = 90;
```

Mohammad Faisal
@mohammadfaisalkhan

Save

Readonly Utility Type

The **Readonly** utility type makes all properties of a given type read-only, meaning that once the properties are assigned values, they cannot be modified. This immutability is enforced by TypeScript's type system, which helps to avoid unintended side effects and maintain the integrity of data.



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

const readonlyPerson: Readonly<Person> = {
  name: 'John',
  age: 30
};

readonlyPerson.name = 'Jane';
// Error: Cannot assign to 'name' because it is a
read-only property.
```

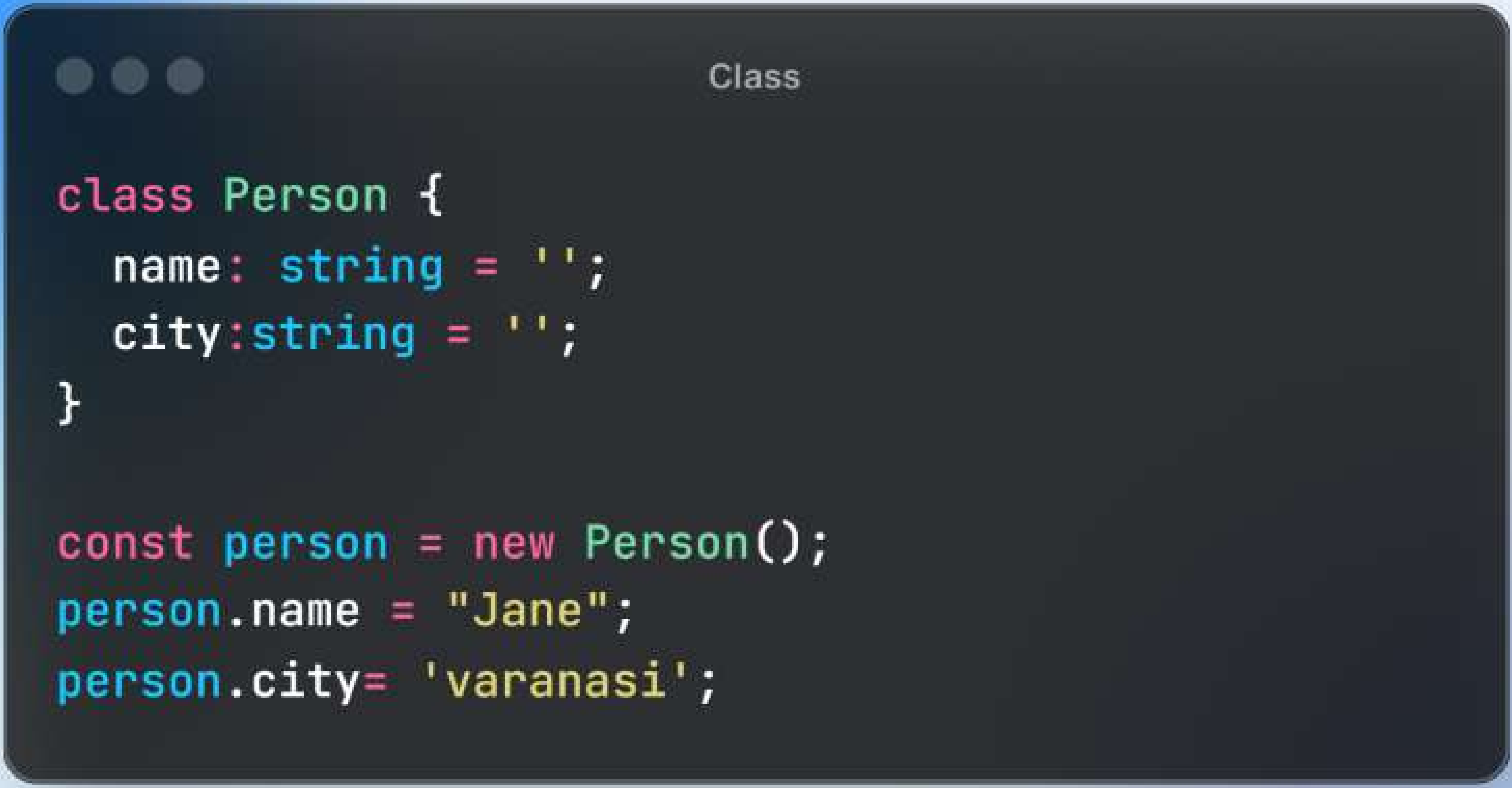
Mohammad Faisal
@mohammadfaisalkhan

Save

OOP features

Classes

TypeScript adds types and visibility modifiers to JavaScript classes. The members of a class (properties & methods) are typed using type annotations, similar to variables.



```
class Person {
  name: string = '';
  city:string = '';
}

const person = new Person();
person.name = "Jane";
person.city= 'varanasi';
```

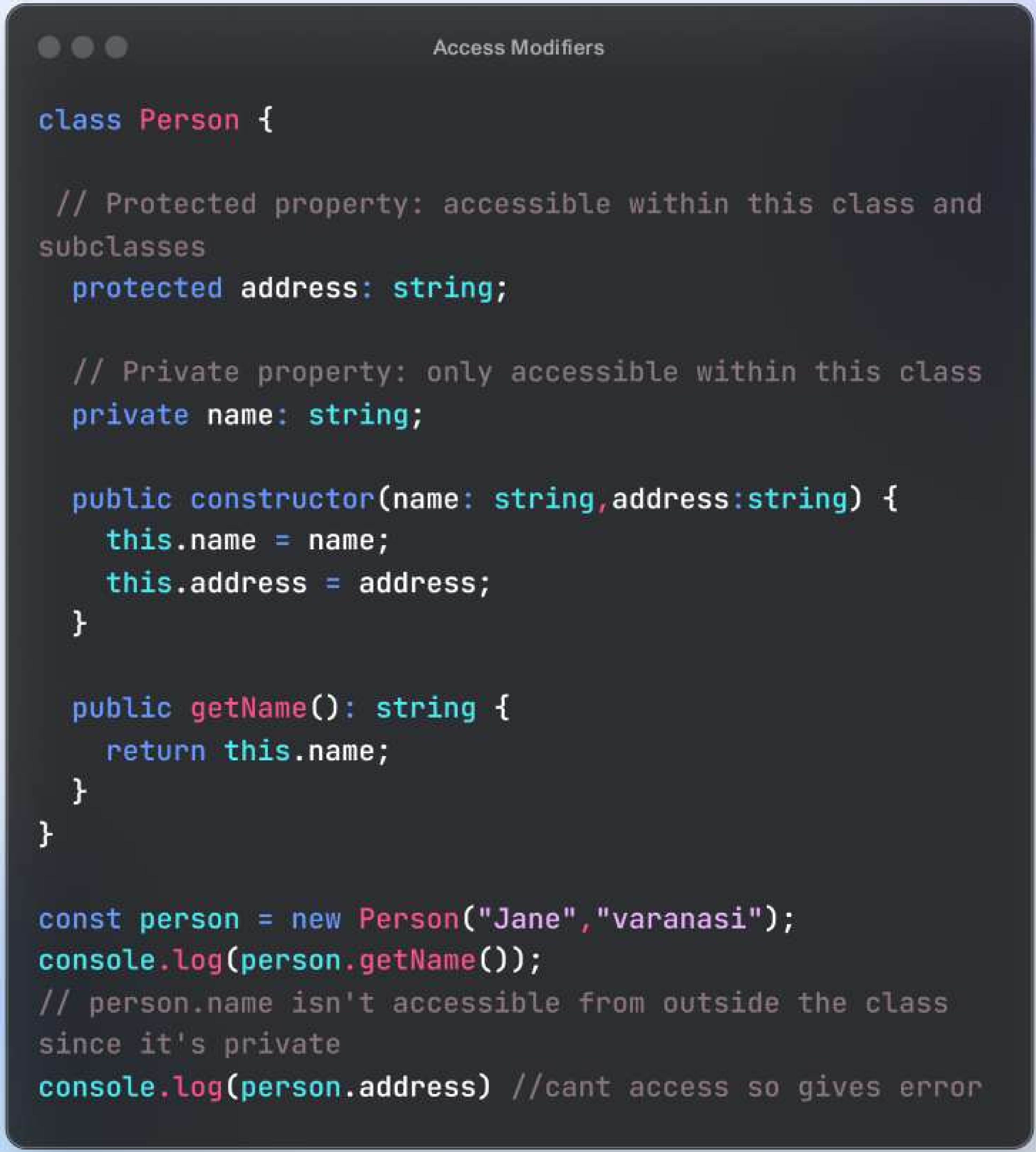
Mohammad Faisal
@mohammadfaisalkhan

Save

Access Modifiers

In TypeScript, access modifiers are keywords used to control the visibility and accessibility of class members (properties and methods). There are three main access/visibility modifiers in TypeScript:

- **public** - (default) Allows access to the class member from anywhere
- **private** - Only allows access to the class member from within the class
- **protected** - Allows access to the class member from itself and any classes that inherit it, which is covered in the inheritance section.


Access Modifiers

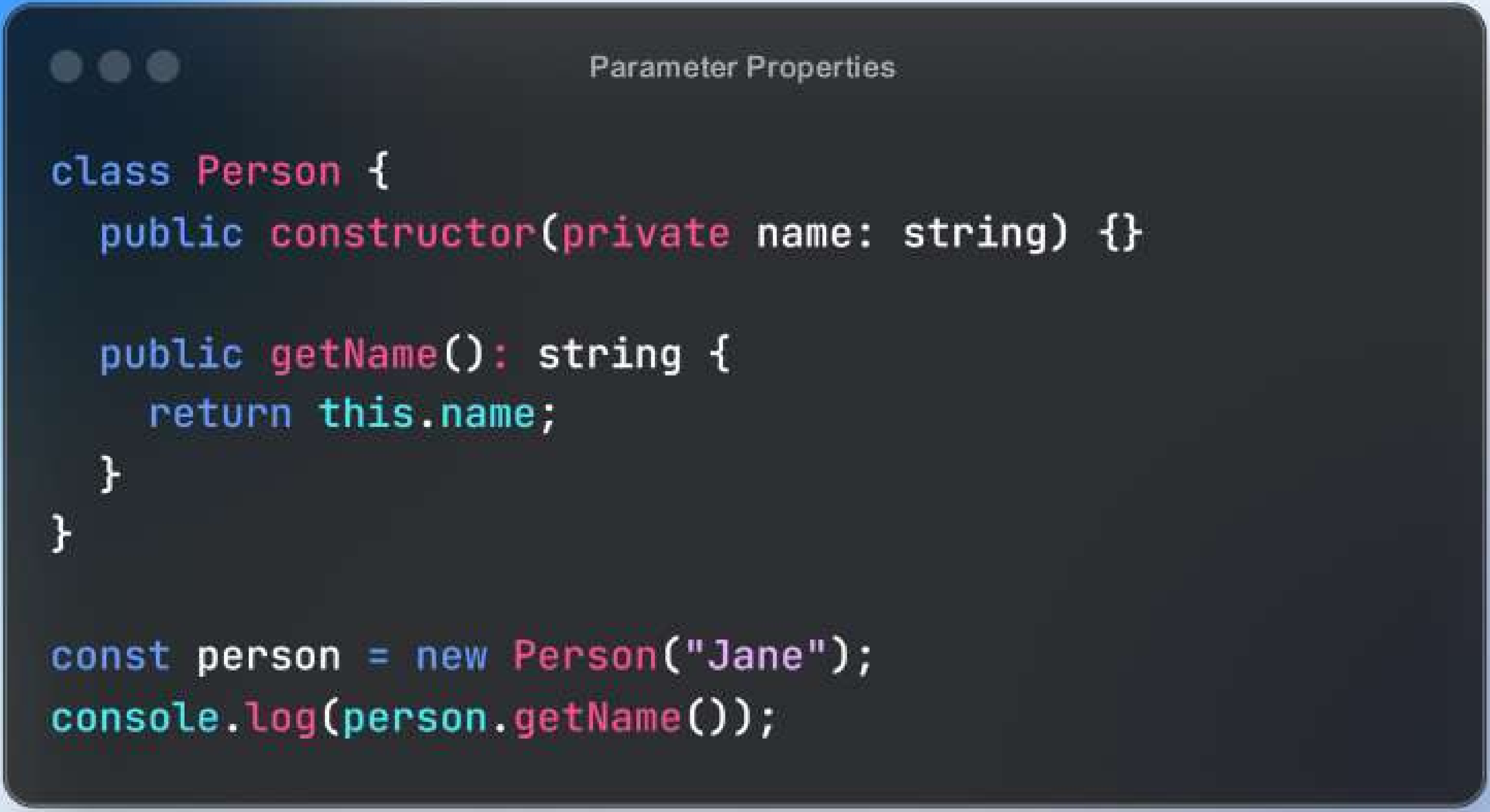
```typescript
class Person {

 // Protected property: accessible within this class and
subclasses
  protected address: string;

  // Private property: only accessible within this class
  private name: string;

  public constructor(name: string,address:string) {
    this.name = name;
    this.address = address;
  }

  public getName(): string {
    return this.name;
  }
}

const person = new Person("Jane","varanasi");
console.log(person.getName());
// person.name isn't accessible from outside the class
since it's private
console.log(person.address) //cant access so gives error
```

Mohammad Faisal
@mohammadfaisalkhan

Save

Parameter Properties

Parameter properties allow you to declare and initialize class properties directly within the constructor parameters by applying visibility modifiers (public, private, or protected). This approach eliminates the need for separate property declarations and assignments inside the constructor body.



```
class Person {
  public constructor(private name: string) {}

  public getName(): string {
    return this.name;
  }
}

const person = new Person("Jane");
console.log(person.getName());
```

Mohammad Faisal
@mohammadfaisalkhan

Save

Readonly

Similar to arrays and object properties, the readonly keyword can prevent class members from being changed.

Readonly

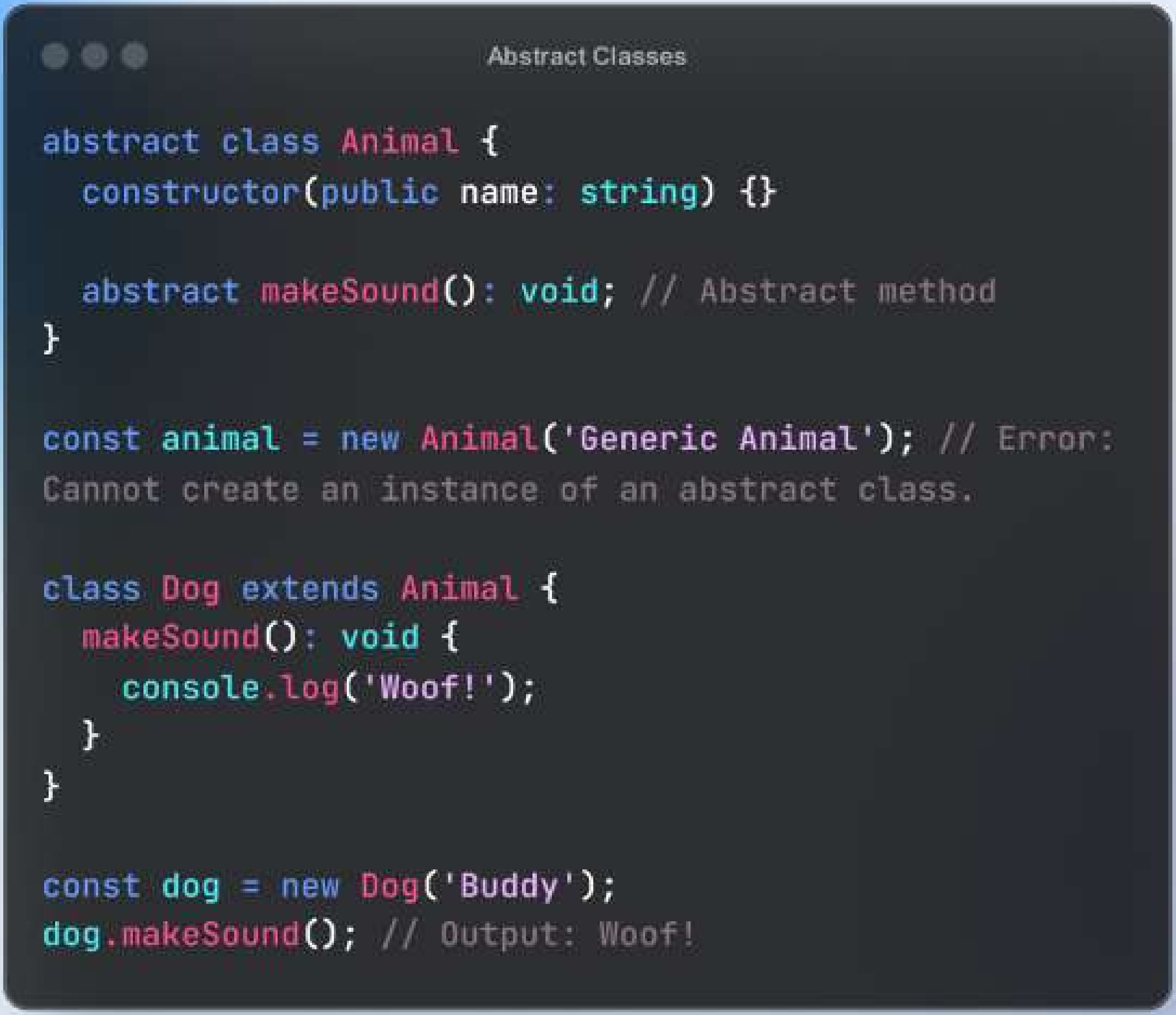
```
class Person {
  private readonly name: string;

  public constructor(name: string) {
    this.name = name;
  }

  public getName(): string {
    return this.name;
  }

  public setName(name: string): void {
    this.name = name; // Error (Not allowed)
  }
}

const person = new Person("Jane");
console.log(person.getName());
```

Mohammad Faisal
@mohammadfaisalkhan

Save

Abstract Classes

It cannot be instantiated directly and serves as a base class for other classes. Defined with the **abstract** keyword, it provides a common interface or partial implementation for its subclasses. Abstract members, marked with **abstract**, must be implemented by derived classes, while non-abstract members can have default implementations.



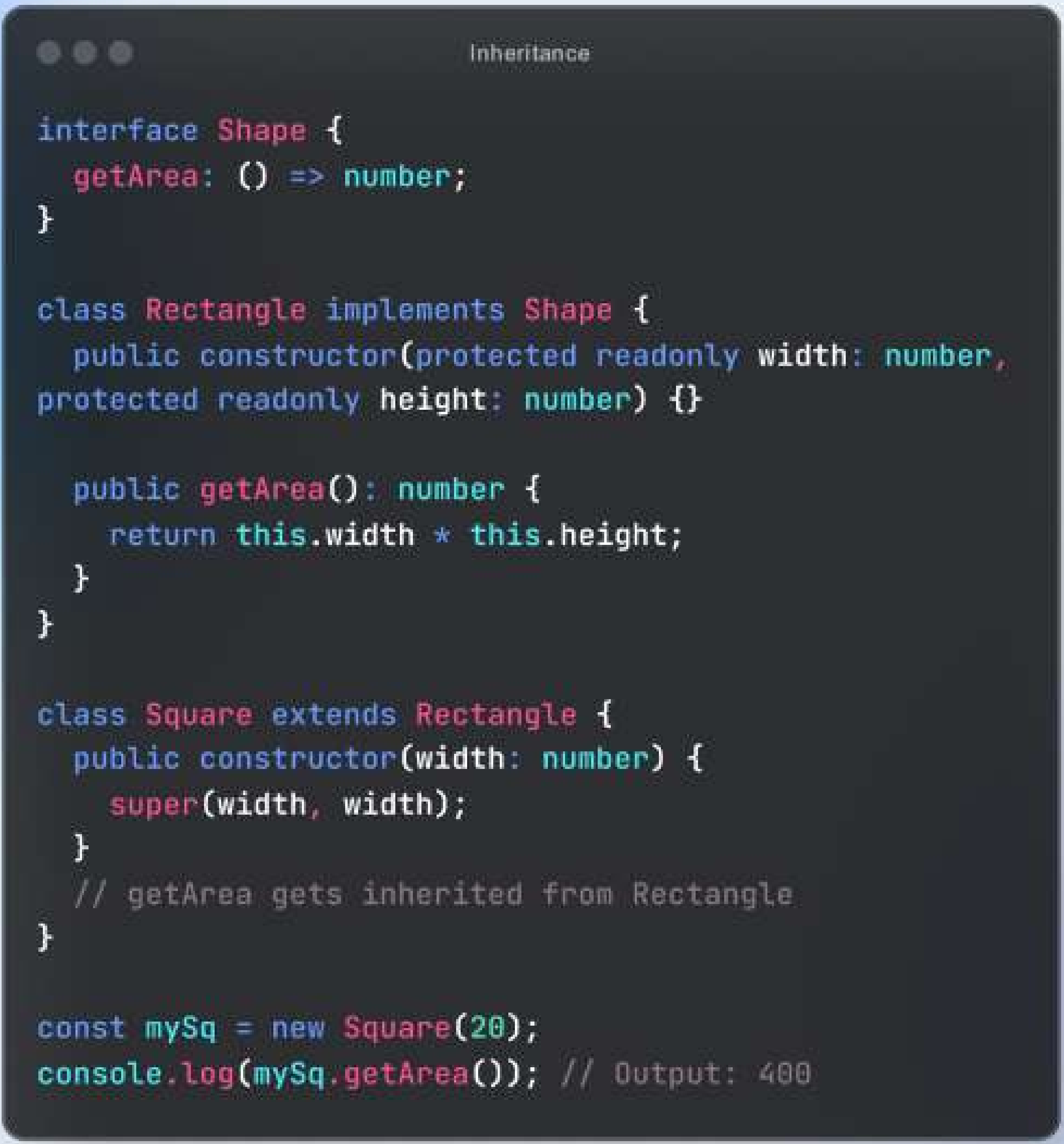
```typescript
abstract class Animal {
  constructor(public name: string) {}

  abstract makeSound(): void; // Abstract method
}

const animal = new Animal('Generic Animal'); // Error:
Cannot create an instance of an abstract class.

class Dog extends Animal {
  makeSound(): void {
    console.log('Woof!');
  }
}

const dog = new Dog('Buddy');
dog.makeSound(); // Output: Woof!
```

Mohammad Faisal
@mohammadfaisalkhan

Save

Inheritance

Classes can extend each other through the extends keyword. A class can only extend one other class. Interfaces can be used to define the type a class must follow through the implements keyword. They are similar to type aliases, except they only apply to object types.


Inheritance

```
interface Shape {
  getArea: () => number;
}

class Rectangle implements Shape {
  public constructor(protected readonly width: number,
protected readonly height: number) {}

  public getArea(): number {
    return this.width * this.height;
  }
}

class Square extends Rectangle {
  public constructor(width: number) {
    super(width, width);
  }
  // getArea gets inherited from Rectangle
}

const mySq = new Square(20);
console.log(mySq.getArea()); // Output: 400
```

Mohammad Faisal
@mohammadfaisalkhan

Save

Follow me for more of those in your LinkedIn feed!

Mohammad Faisal
@mohammadfaisalkhan

Mohammad Faisal
@mohammadfaisalkhan

Save