Introduction to TypeScript

Benoit Ruiz

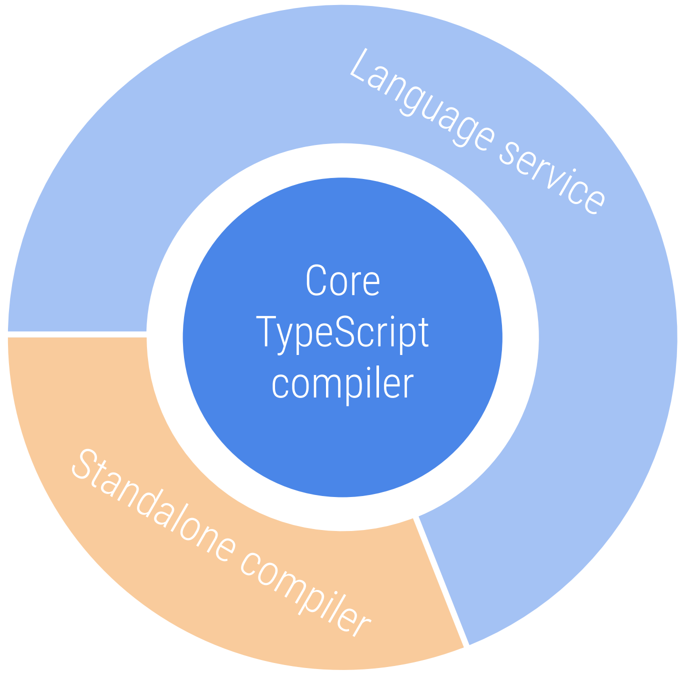
Table of contents

What is TypeScript?

What is TypeScript?

- An Open Source programming language that compiles to JavaScript
- Co-created by Anders Hejlsberg at Microsoft
- The first public release was on October 2012 (v0.8)
- A superset of JavaScript, adding optional static types

What is TypeScript?



Architectural overview

Core TypeScript compiler

Compiles TypeScript into JavaScript (*transpilation*)

Language service

Used by editors: completions, help tips, code formatting…

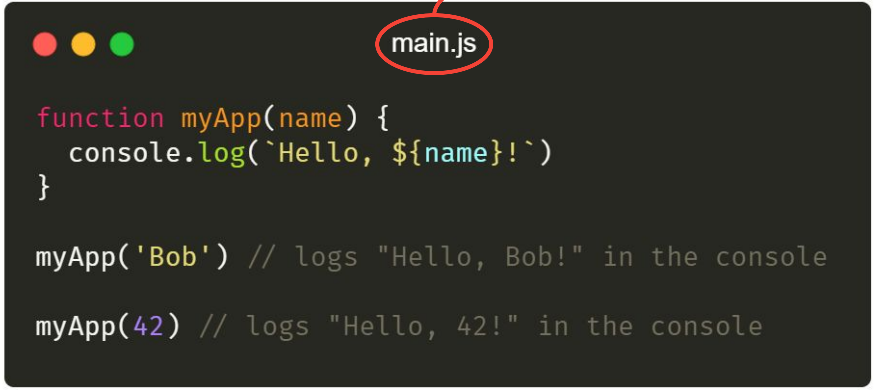
Standalone compiler

`tsc` program

A superset of JavaScript

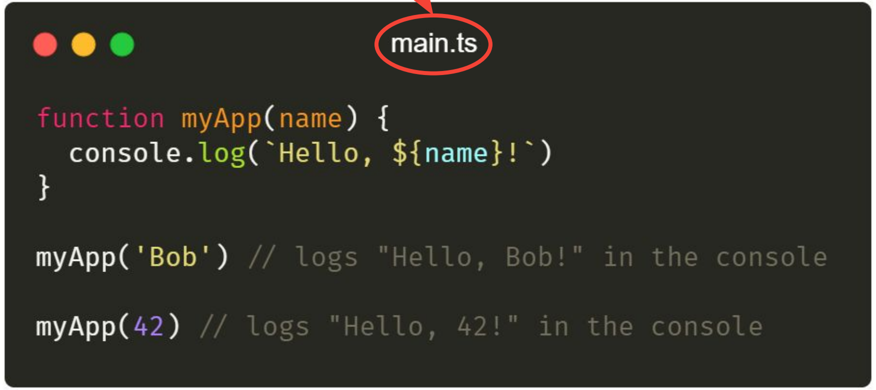
A JavaScript program is a valid TypeScript program*.

Rename file extension from .js to .ts



main.js

```
function myApp(name) {
  console.log(`Hello, ${name}!`)
}

myApp('Bob') // logs "Hello, Bob!" in the console

myApp(42) // logs "Hello, 42!" in the console
```



main.ts

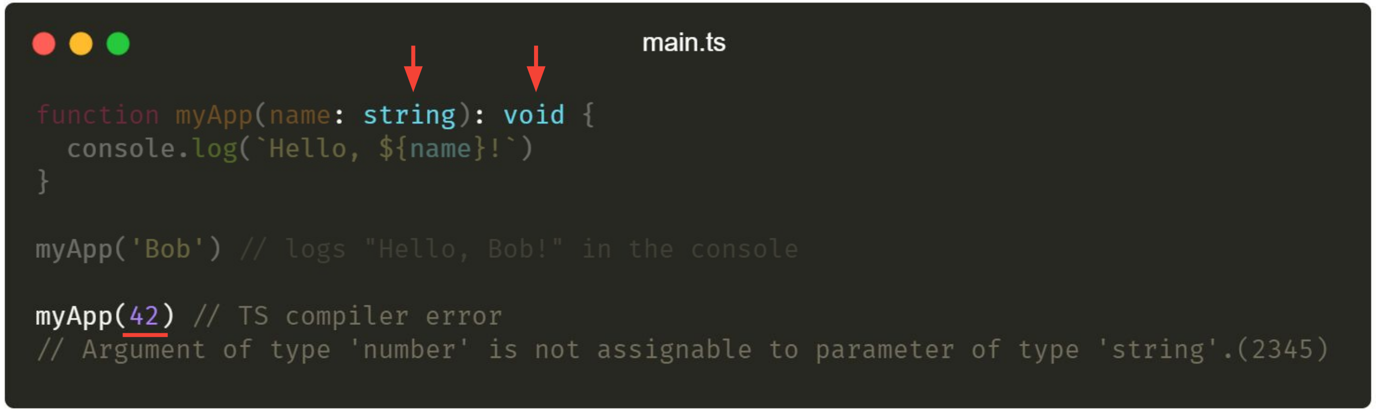
```
function myApp(name) {
  console.log(`Hello, ${name}!`)
}

myApp('Bob') // logs "Hello, Bob!" in the console

myApp(42) // logs "Hello, 42!" in the console
```

Valid TypeScript program!

** As long as “strict” rules are not enabled in the tsconfig.json configuration file. More on that later.*

A superset of JavaScript

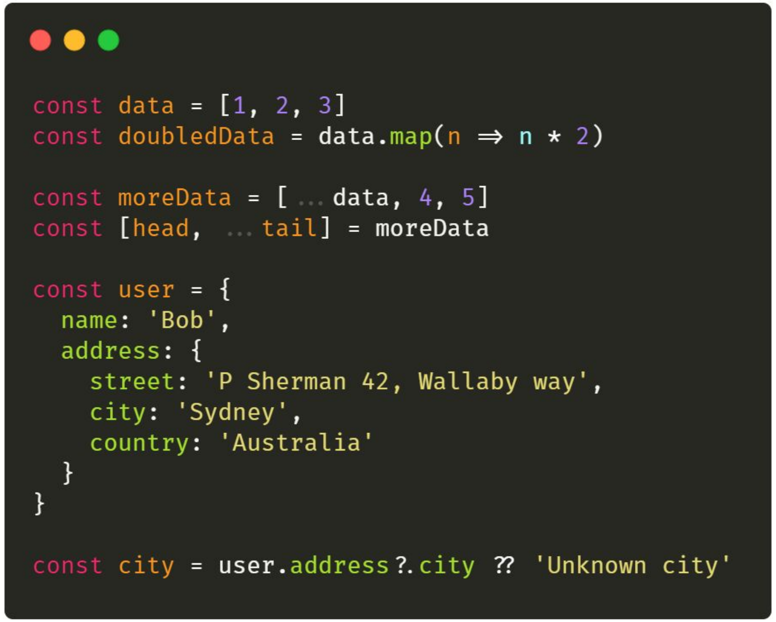
TypeScript adds (optional) static types on top of JavaScript.



```
function myApp(name: string): void {
  console.log(`Hello, ${name}!`)
}

myApp('Bob') // logs "Hello, Bob!" in the console

myApp(42) // TS compiler error
// Argument of type 'number' is not assignable to parameter of type 'string'.(2345)
```

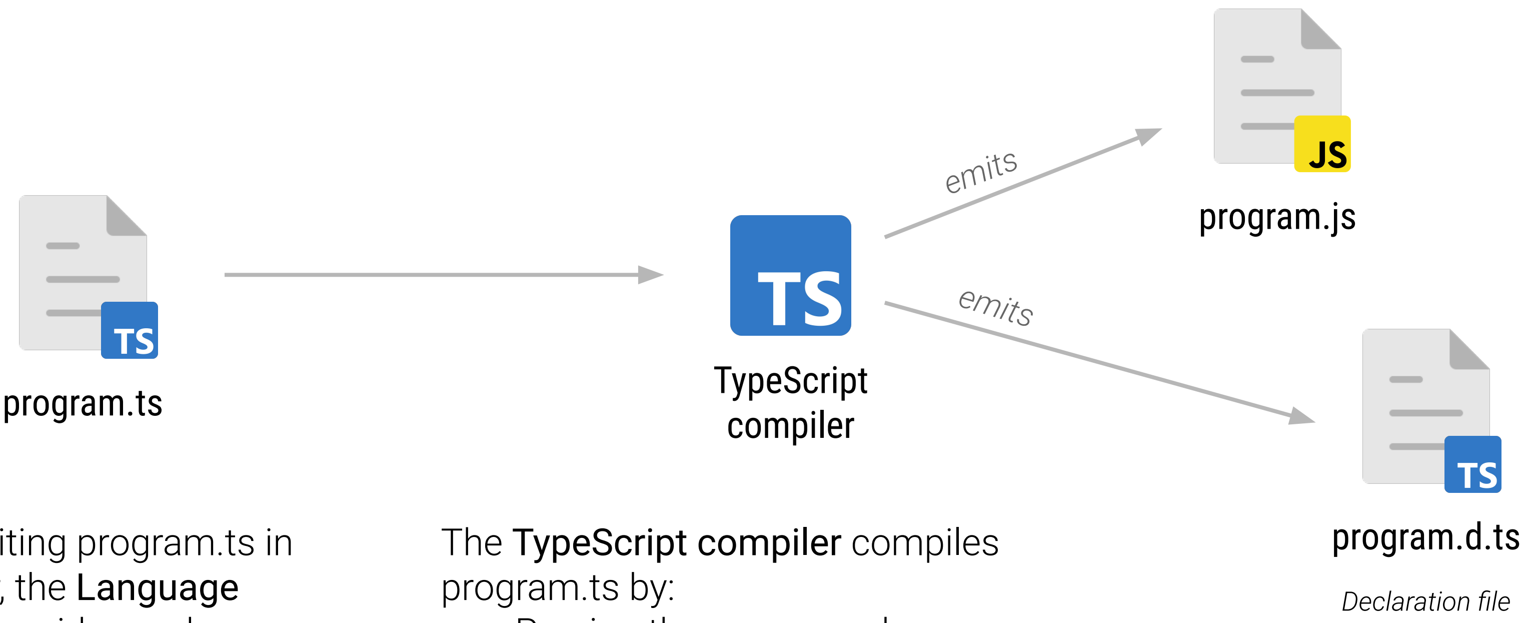
A superset of JavaScript

It supports the latest stable features of EcmaScript specification, e.g. arrow functions, optional chaining operator, rest and spread syntax…



```javascript
const data = [1, 2, 3]
const doubledData = data.map(n => n * 2)

const moreData = [...data, 4, 5]
const [head, ...tail] = moreData

const user = {
  name: 'Bob',
  address: {
    street: 'P Sherman 42, Wallaby way',
    city: 'Sydney',
    country: 'Australia'
  }
}

const city = user.address?.city ?? 'Unknown city'
```

Overview



While writing program.ts in an editor, the **Language service** provides code completions, help tips and code formatting.

The **TypeScript compiler** compiles program.ts by:

- Parsing the source code
- **Type checking** it
- Emitting output files

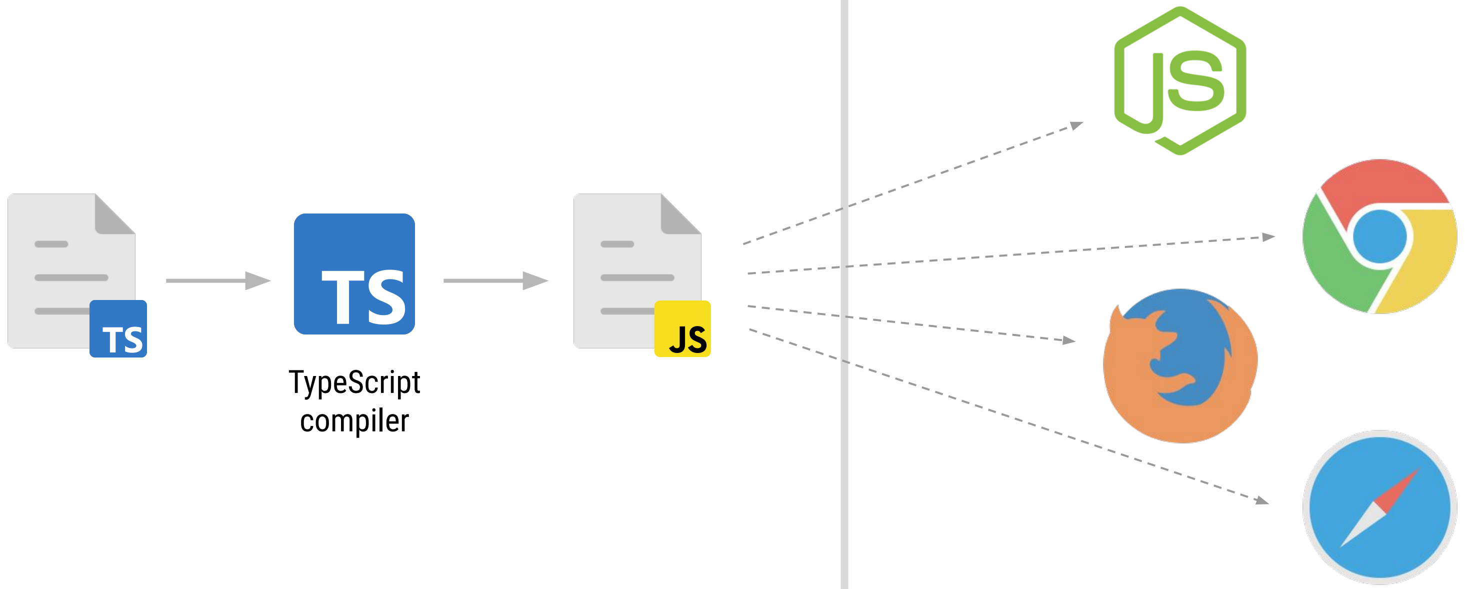
Overview

🏗 Compile time

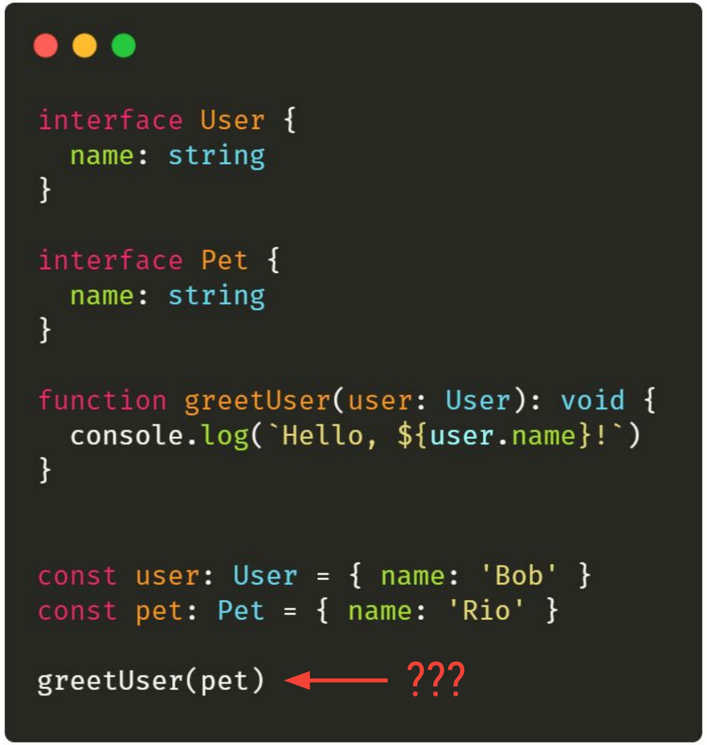
Static typing.

⚙ Runtime

Execution of the program. Dynamic typing.



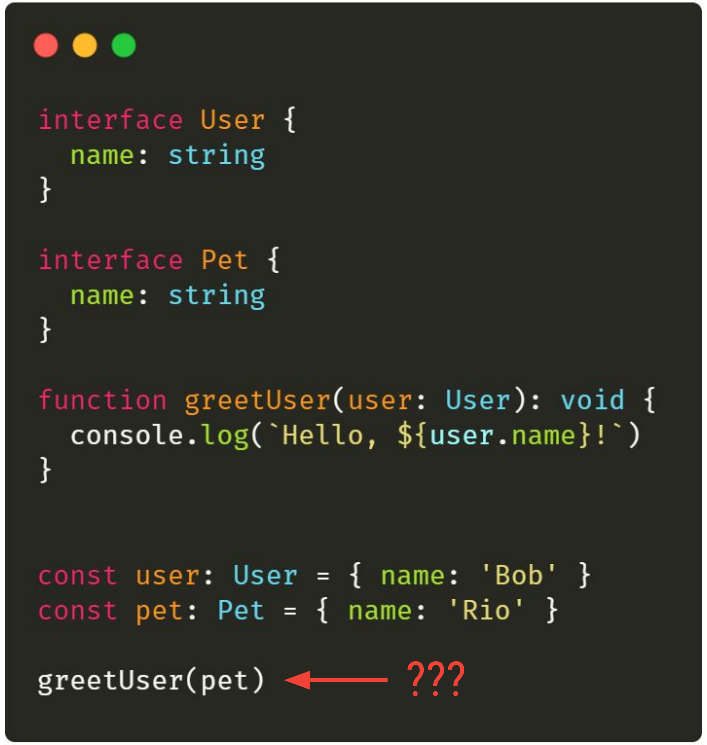
Type compatibility



```
interface User {
  name: string
}

interface Pet {
  name: string
}

function greetUser(user: User): void {
  console.log(`Hello, ${user.name}!`)
}


const user: User = { name: 'Bob' }
const pet: Pet = { name: 'Rio' }

greetUser(pet)
```

What is going to happen?

Type compatibility



```
interface User {
  name: string
}

interface Pet {
  name: string
}

function greetUser(user: User): void {
  console.log(`Hello, ${user.name}!`)
}


const user: User = { name: 'Bob' }
const pet: Pet = { name: 'Rio' }

greetUser(pet)
```

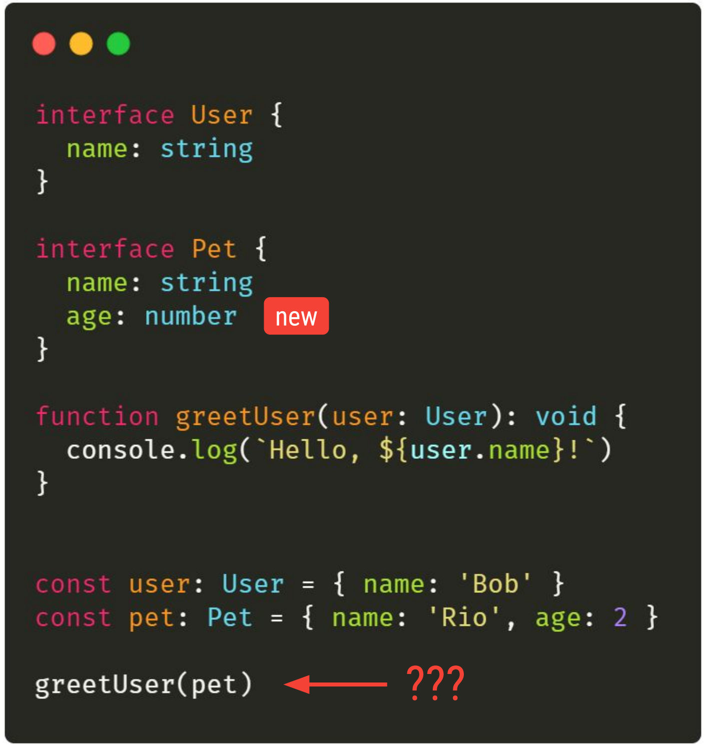
What is going to happen?

No error!

TypeScript is based on **structural** typing (in contrast with **nominal** typing).

“If it looks like a duck, and quacks like a duck, then it's a duck”. 🦆

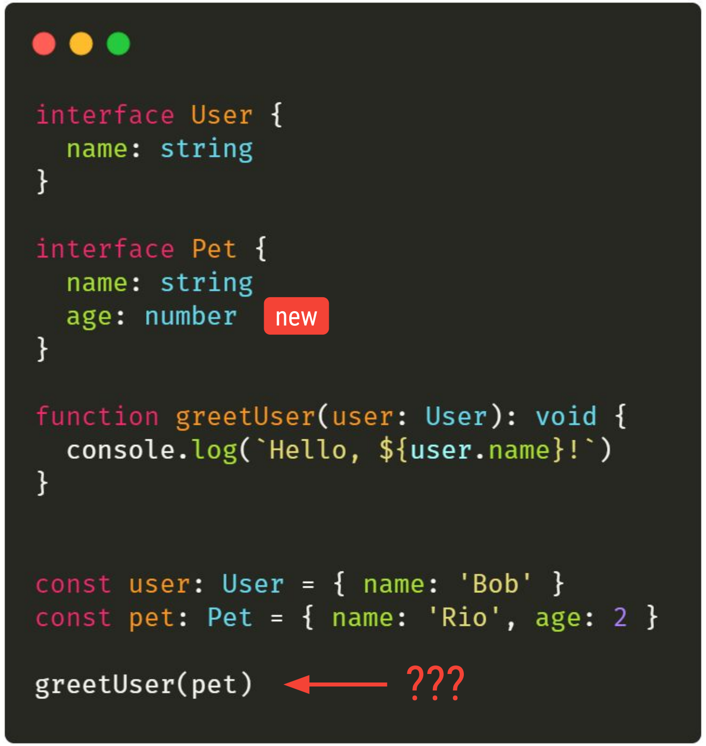
Type compatibility



```
interface User {
  name: string
}

interface Pet {
  name: string
  age: number  new
}

function greetUser(user: User): void {
  console.log(`Hello, ${user.name}!`)
}


const user: User = { name: 'Bob' }
const pet: Pet = { name: 'Rio', age: 2 }

greetUser(pet)  ←—— ???
```

What is going to happen?

Type compatibility



```
interface User {
  name: string
}

interface Pet {
  name: string
  age: number   new
}

function greetUser(user: User): void {
  console.log(`Hello, ${user.name}!`)
}


const user: User = { name: 'Bob' }
const pet: Pet = { name: 'Rio', age: 2 }

greetUser(pet)  ←— ???
```

What is going to happen?

Still no error!

The compiler only checks that **at least** the properties required are present and match the types required.

However, there are some cases where TypeScript would throw a compiler error, cf. excess property checks.

Type compatibility

Why is TypeScript using structural typing instead of nominal typing like in Java or C#?

Type system was designed based on how JavaScript code is **typically written**:

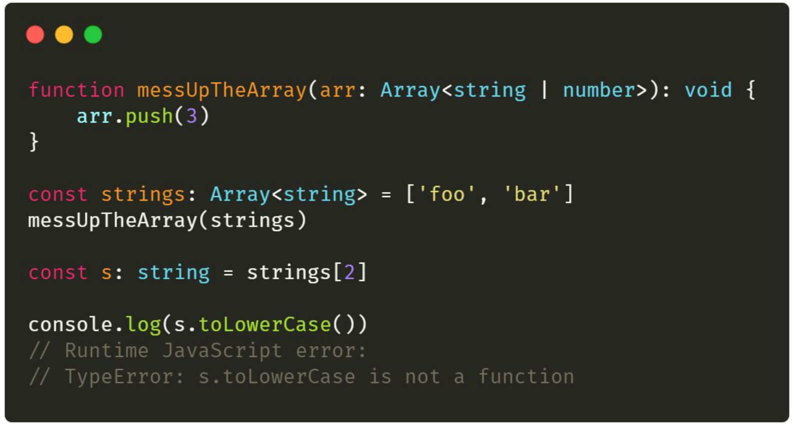
- JavaScript widely uses **anonymous objects** (function expressions, object literals…) `const f = () ⇒ 42` `const foo = { name: 'Bob' }`
- It's more natural to **represent relationships found in JS libraries** using a structural type system instead of a nominal one.

Soundness

A **sound** type system implies that all type-checked programs are correct.

Simply put, if the compiler states that a variable has a particular type, then it definitely has that type at **runtime**.

TypeScript type system is **NOT sound**.



```typescript
function messUpTheArray(arr: Array<string | number>): void {
    arr.push(3)
}

const strings: Array<string> = ['foo', 'bar']
messUpTheArray(strings)

const s: string = strings[2]

console.log(s.toLowerCase())
// Runtime JavaScript error:
// TypeError: s.toLowerCase is not a function
```

Why should we use it?

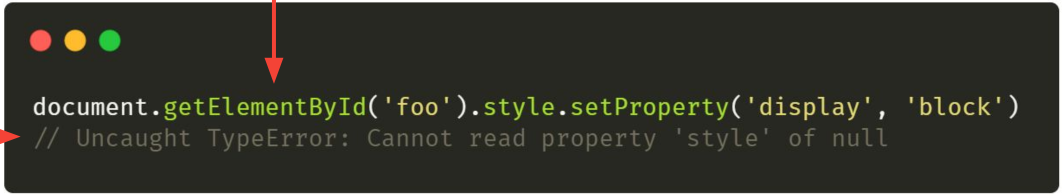
Why should we use it?

It helps us **avoid bugs and runtime errors** commonly encountered by JavaScript developers.

It is highly recommended to enable the **strict mode** for increased safety.

In strict mode, TS compiler would fail with:
Object is possibly 'null'.(2531)

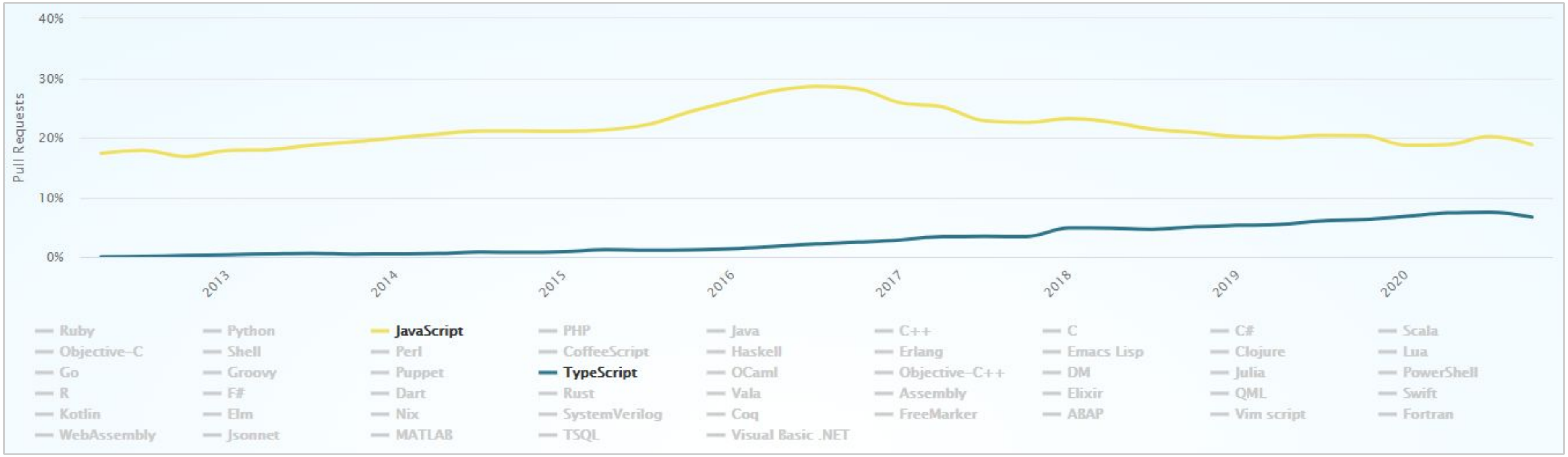
Running this JS code would result in a runtime error.



```
document.getElementById('foo').style.setProperty('display', 'block')
// Uncaught TypeError: Cannot read property 'style' of null
```

Why should we use it?

Its **popularity** is in constant growth since its first release.

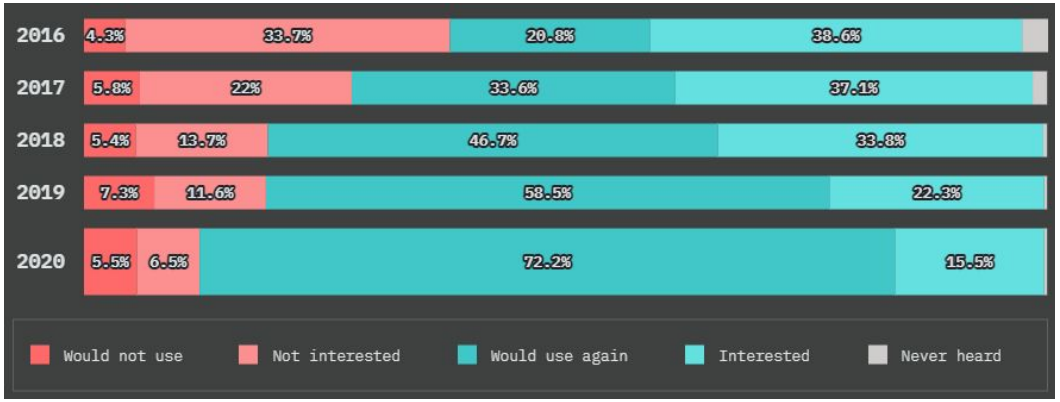


In 2020, TypeScript was 7th in Pull-Requests activity on GitHub.

Why should we use it?

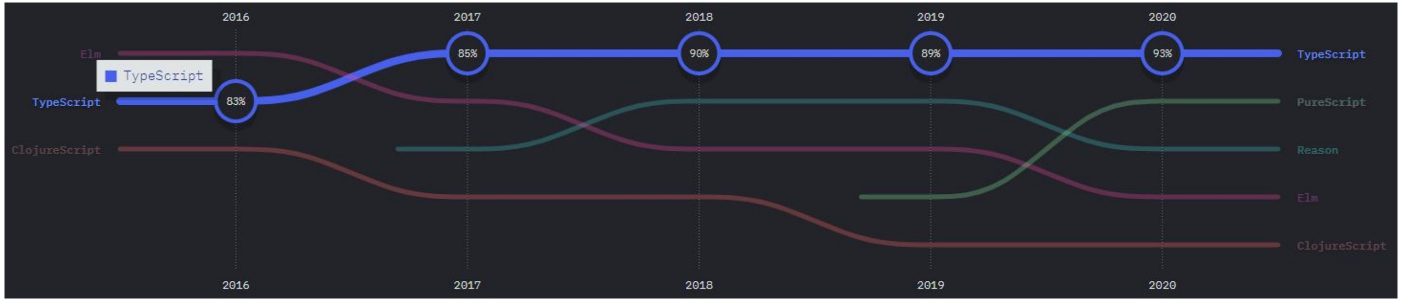
Its **popularity** is in constant growth since its first release.

Experience with TypeScript over time



State of JS 2020 survey

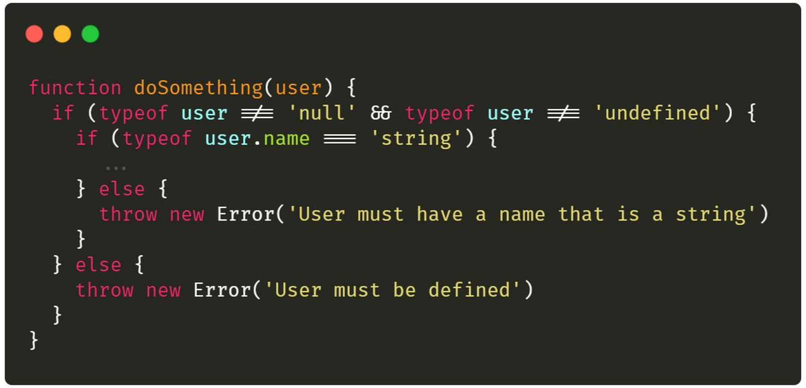
Satisfaction over time



Why should we use it?

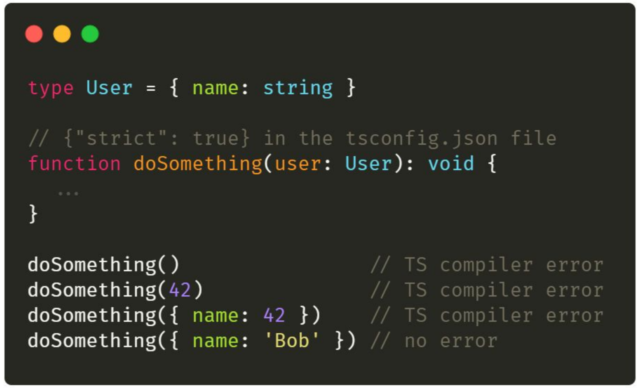
Overall, it increases developers **productivity**.

For internal* code, a lot of unit tests can be avoided thanks to the **type system**.



```
function doSomething(user) {
  if (typeof user !== 'null' && typeof user !== 'undefined') {
    if (typeof user.name === 'string') {
      ...
    } else {
      throw new Error('User must have a name that is a string')
    }
  } else {
    throw new Error('User must be defined')
  }
}
```

There is no need for runtime checks and unit tests, the compiler catches the error cases!



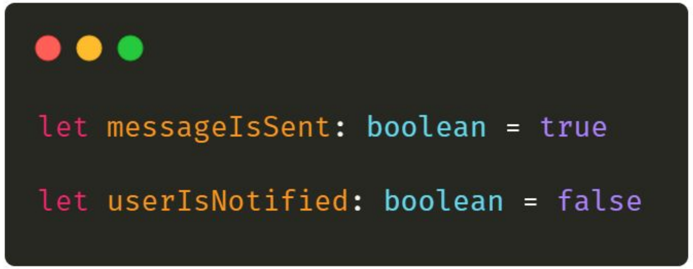
```
type User = { name: string }

// {"strict": true} in the tsconfig.json file
function doSomething(user: User): void {
  ...
}

doSomething()                    // TS compiler error
doSomething(42)                  // TS compiler error
doSomething({ name: 42 })        // TS compiler error
doSomething({ name: 'Bob' }) // no error
```

** Code written by the developer, that is not coming from external dependencies such as libraries, DOM, APIs...*

Language features

Basic types

Boolean

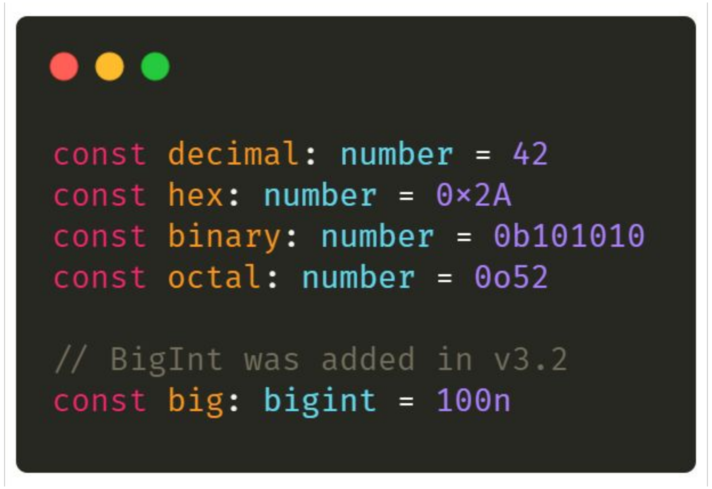


```
let messageIsSent: boolean = true

let userIsNotified: boolean = false
```

The `Boolean` type (in contrast with `boolean` used here) should never be used as a type. Same goes for `String`, `Number`, `BigInt`, `Symbol` and `Object`.

Basic types

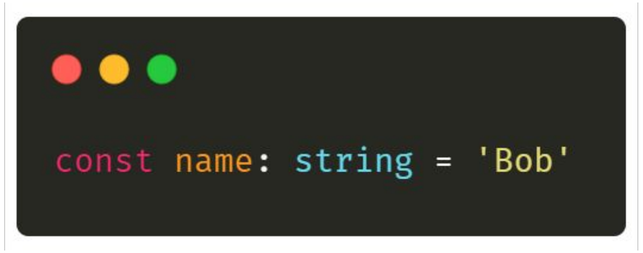
Number
and
BigInt



```
const decimal: number = 42
const hex: number = 0×2A
const binary: number = 0b101010
const octal: number = 0o52

// BigInt was added in v3.2
const big: bigint = 100n
```

Basic types

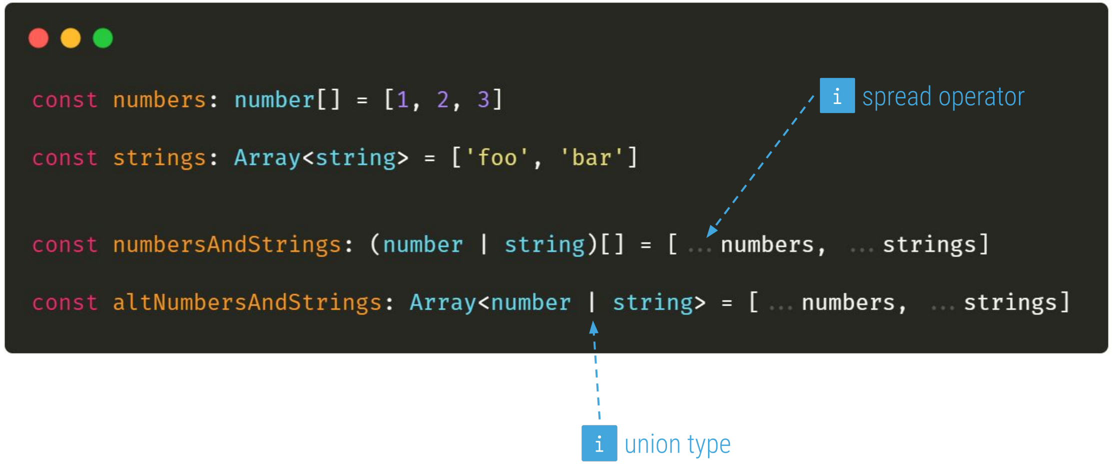
String



```
const name: string = 'Bob'
```

Basic types

Array



```
const numbers: number[] = [1, 2, 3]
const strings: Array<string> = ['foo', 'bar']

const numbersAndStrings: (number | string)[] = [...numbers, ...strings]
const altNumbersAndStrings: Array<number | string> = [...numbers, ...strings]
```

Basic types

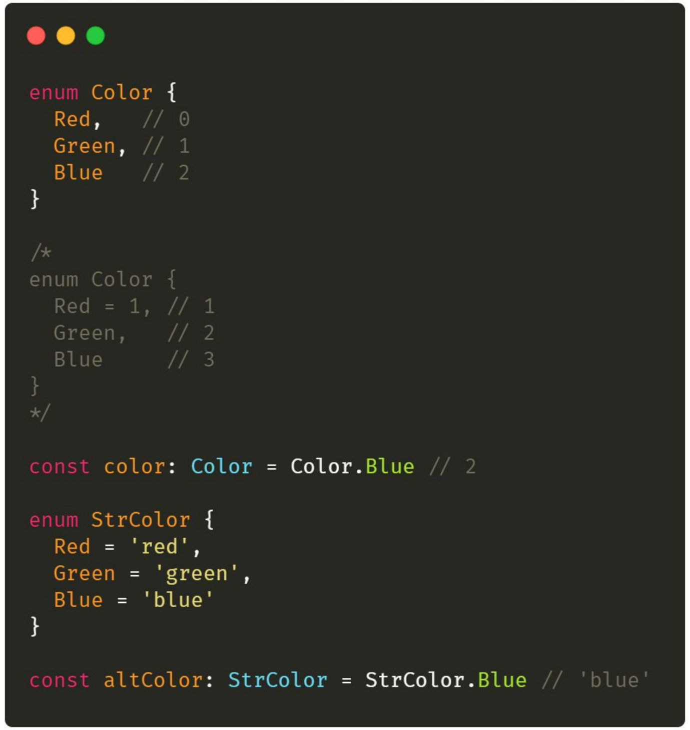
Tuple

```
const user: [string, number] = ['Bob', 25]

// Labeled tuples were introduced in v4.0
const altUser: [name: string, age: number] = ['Bob', 25]


console.log(user[0].toLowerCase()) // no error, logs "bob"
console.log(user[1].toFixed(2))    // no error, logs "25.00"

user[2] = 'some value' // TS compiler error
// Tuple type '[string, number]' of length '2' has no element at index '2'.
```

An array with a fixed number of elements.

Basic types

Enum

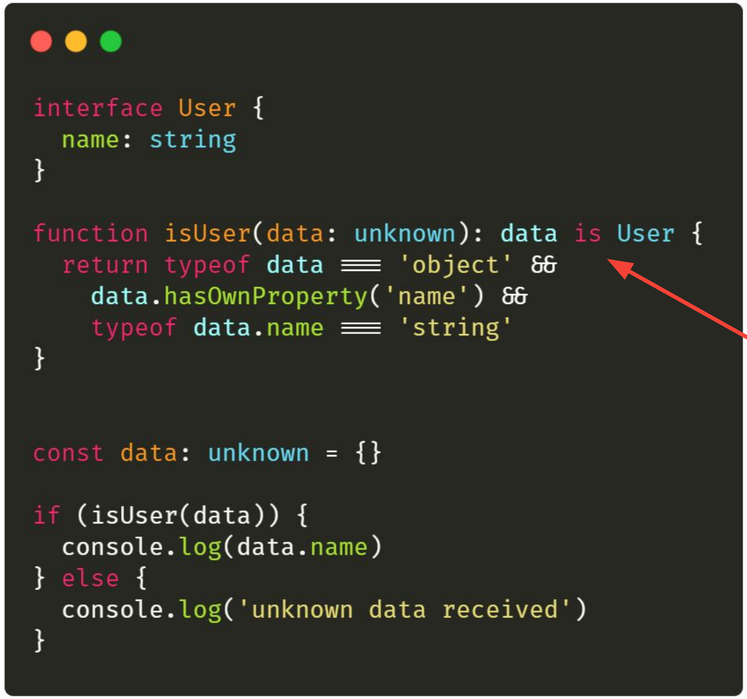


```
enum Color {
  Red,    // 0
  Green, // 1
  Blue   // 2
}

/*
enum Color {
  Red = 1, // 1
  Green,   // 2
  Blue     // 3
}
*/

const color: Color = Color.Blue // 2

enum StrColor {
  Red = 'red',
  Green = 'green',
  Blue = 'blue'
}

const altColor: StrColor = StrColor.Blue // 'blue'
```

Useful to assign human-readable labels to “raw” values such as numbers and strings.

Basic types

Unknown

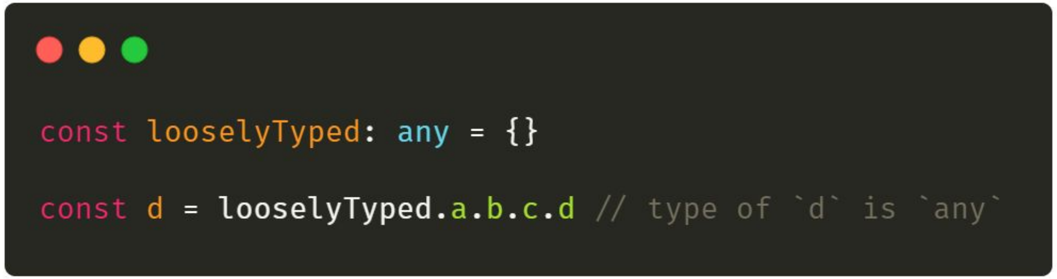


```
interface User {
  name: string
}

function isUser(data: unknown): data is User {
  return typeof data === 'object' &&
    data.hasOwnProperty('name') &&
    typeof data.name === 'string'
}


const data: unknown = {}

if (isUser(data)) {
  console.log(data.name)
} else {
  console.log('unknown data received')
}
```

This is a type predicate. It is used to inform the compiler that "parameter `data` has the type `User` if the function returns `true`, otherwise it keeps the `unknown` type".

Basic types

Any



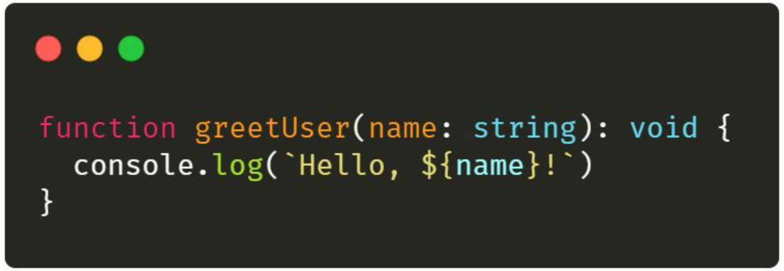
```
const looselyTyped: any = {}

const d = looselyTyped.a.b.c.d // type of `d` is `any`
```

Avoid at all cost!

Prefer using **unknown** when you are unsure about the type of a value.

Basic types

Void

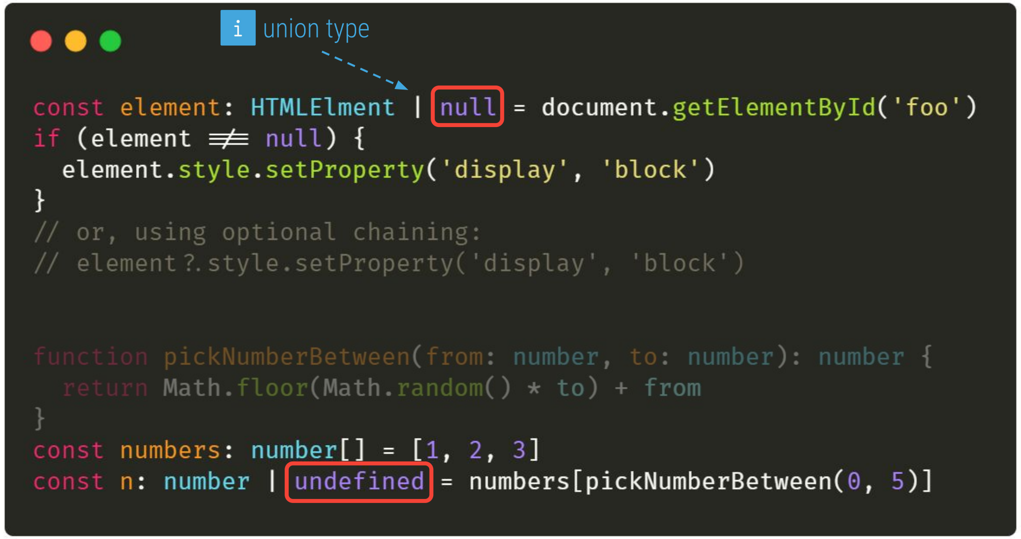


```
function greetUser(name: string): void {
  console.log(`Hello, ${name}!`)
}
```

Use **`void`** as the return type of a function that doesn’t return any value.

Basic types

Null
and
Undefined

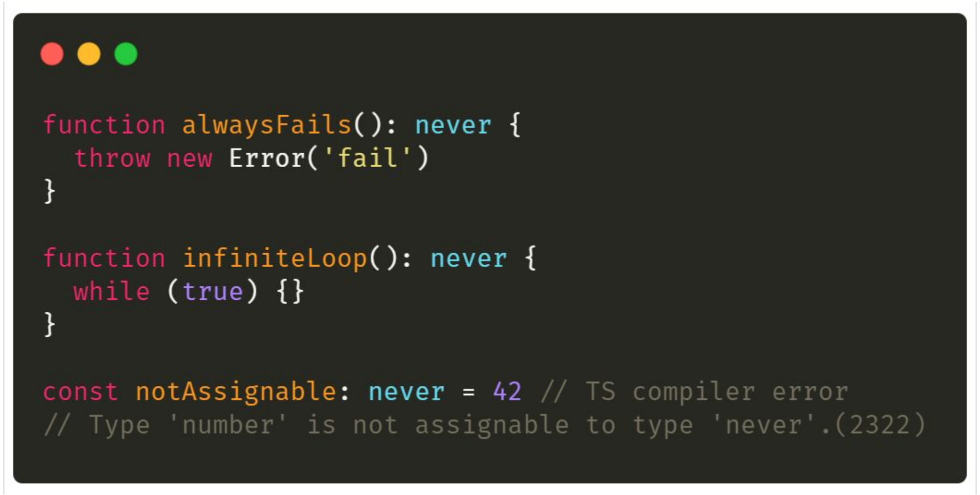


```
const element: HTMLElment | null = document.getElementById('foo')
if (element !== null) {
  element.style.setProperty('display', 'block')
}
// or, using optional chaining:
// element?.style.setProperty('display', 'block')


function pickNumberBetween(from: number, to: number): number {
  return Math.floor(Math.random() * to) + from
}
const numbers: number[] = [1, 2, 3]
const n: number | undefined = numbers[pickNumberBetween(0, 5)]
```

Strict mode (more specifically, the *strictNullChecks* flag) must be enabled so `null` and `undefined` are not assignable to any type.
Enabling this flag is highly **recommended** to avoid many common mistakes.

Basic types

Never



```
function alwaysFails(): never {
  throw new Error('fail')
}

function infiniteLoop(): never {
  while (true) {}
}

const notAssignable: never = 42 // TS compiler error
// Type 'number' is not assignable to type 'never'.(2322)
```

Nothing is assignable to **`never`**.

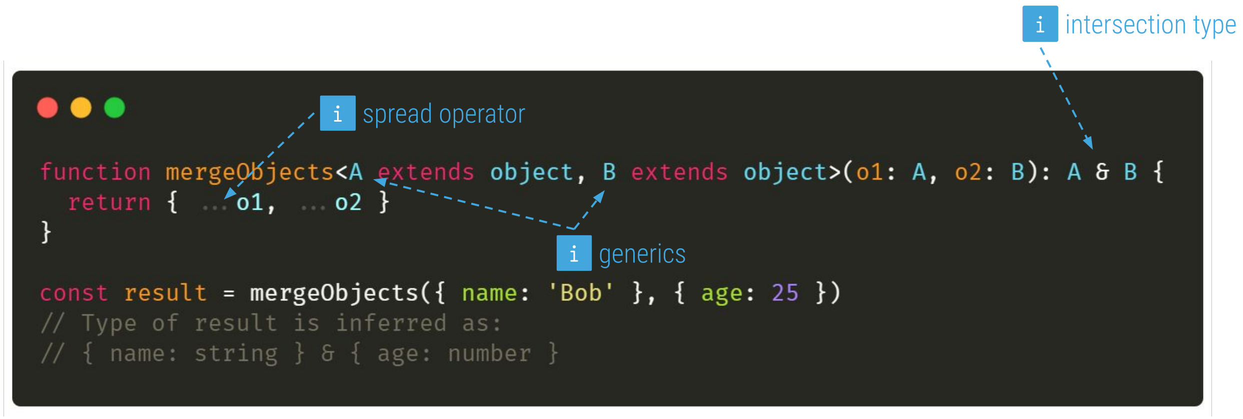
The **`never`** type really shines when writing conditional types.

Basic types

Object

It represents non-primitive types, i.e. anything that is not `boolean`, `number`, `bigint`, `string`, `symbol`, `null` or `undefined`.

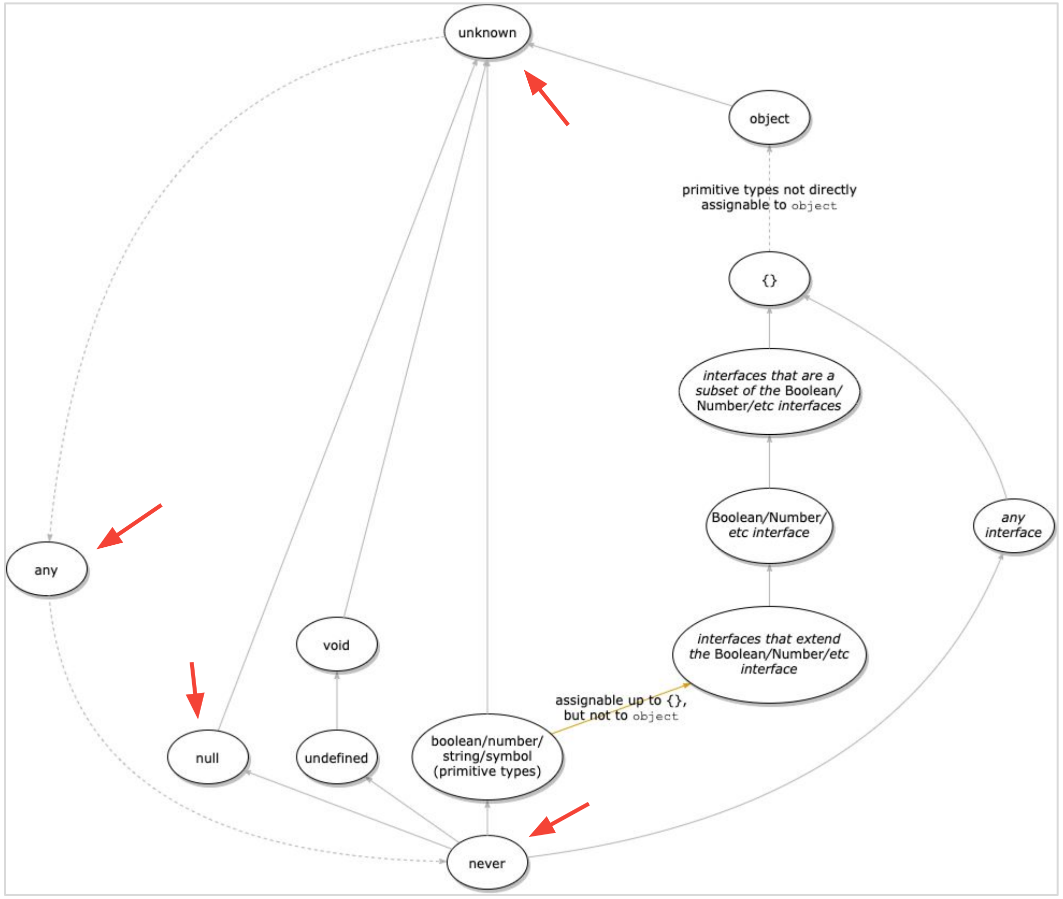
Generally, you won't need to use `object`, you will most likely define interfaces or record types instead (e.g. `User`).



```
function mergeObjects<A extends object, B extends object>(o1: A, o2: B): A & B {
  return { ...o1, ...o2 }
}

const result = mergeObjects({ name: 'Bob' }, { age: 25 })
// Type of result is inferred as:
// { name: string } & { age: number }
```

We'll talk about generics, type inference and intersection types later on.

Types hierarchy



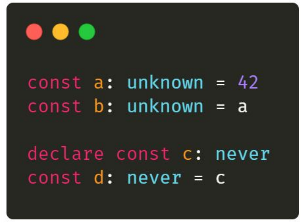
unknown
object
primitive types not directly assignable to object
{}
interfaces that are a subset of the Boolean/Number/etc interfaces
Boolean/Number/etc interface
interfaces that extend the Boolean/Number/etc interface
any interface
any
void
null
undefined
boolean/number/string/symbol (primitive types)
assignable up to {}, but not to object
never

Anything is assignable to **unknown** ("top type"). **unknown** is only assignable to itself and **any**.

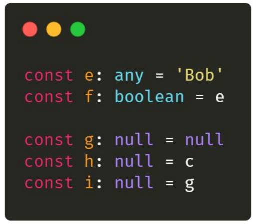
Nothing is assignable to **never** ("bottom type"). **never** is only assignable to itself and **any**.

Anything is assignable to **any**, and **any** is assignable to anything. Avoid this type as much as possible (prefer using **unknown**).

Nothing besides **null**, **any** and **never** is assignable to **null** ("unit type"). **null** is assignable to itself.



```
const a: unknown = 42
const b: unknown = a

declare const c: never
const d: never = c
```

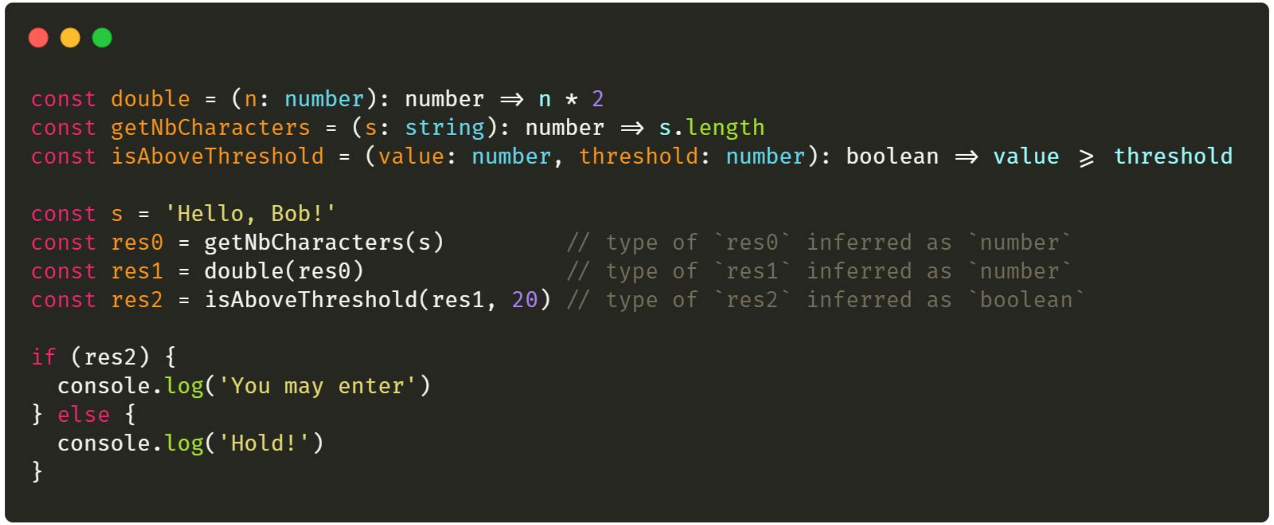


```
const e: any = 'Bob'
const f: boolean = e

const g: null = null
const h: null = c
const i: null = g
```

Type inference

TypeScript is able to **infer** (i.e. “guess”) the resulting type of an expression.

This allows the developer to see **how TypeScript understands** his code, and he *can* **omit** the types of some variables and return values as well.



```
const double = (n: number): number => n * 2
const getNbCharacters = (s: string): number => s.length
const isAboveThreshold = (value: number, threshold: number): boolean => value >= threshold

const s = 'Hello, Bob!'
const res0 = getNbCharacters(s)         // type of `res0` inferred as `number`
const res1 = double(res0)               // type of `res1` inferred as `number`
const res2 = isAboveThreshold(res1, 20) // type of `res2` inferred as `boolean`

if (res2) {
  console.log('You may enter')
} else {
  console.log('Hold!')
}
```

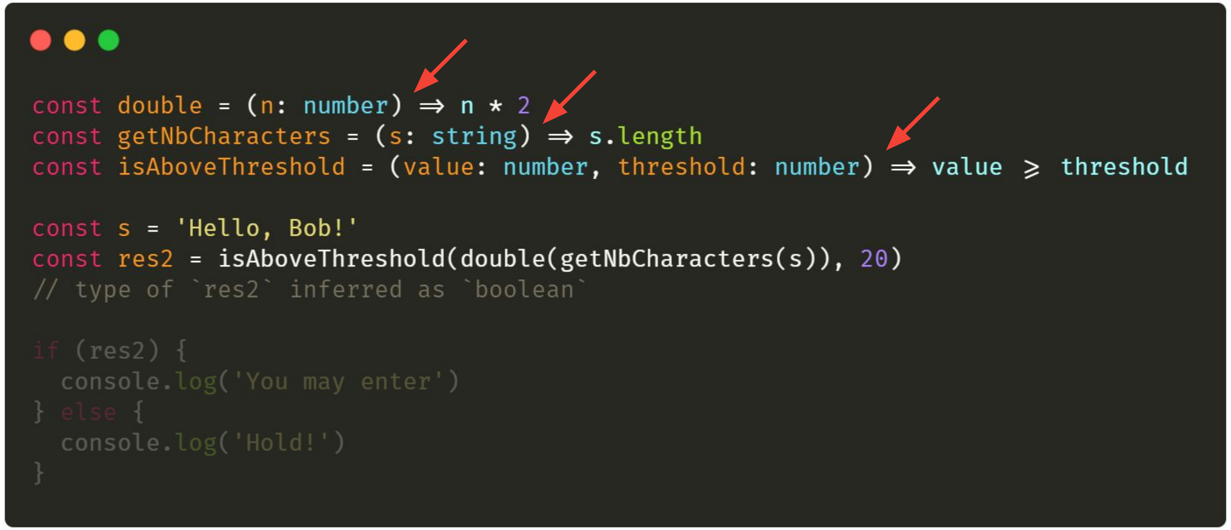
i

TIP: in your editor (e.g. VS Code), you can use ctrl + mouse cursor (or cmd + mouse cursor on Mac) to see the inferred type of an expression.

Type inference

TypeScript is able to **infer** (i.e. “guess”) the resulting type of an expression.

This allows the developer to see **how TypeScript understands** his code, and he *can* **omit** the types of some variables and return values as well.



```
const double = (n: number) ⇒ n * 2
const getNbCharacters = (s: string) ⇒ s.length
const isAboveThreshold = (value: number, threshold: number) ⇒ value ≥ threshold

const s = 'Hello, Bob!'
const res2 = isAboveThreshold(double(getNbCharacters(s)), 20)
// type of `res2` inferred as `boolean`

if (res2) {
  console.log('You may enter')
} else {
  console.log('Hold!')
}
```

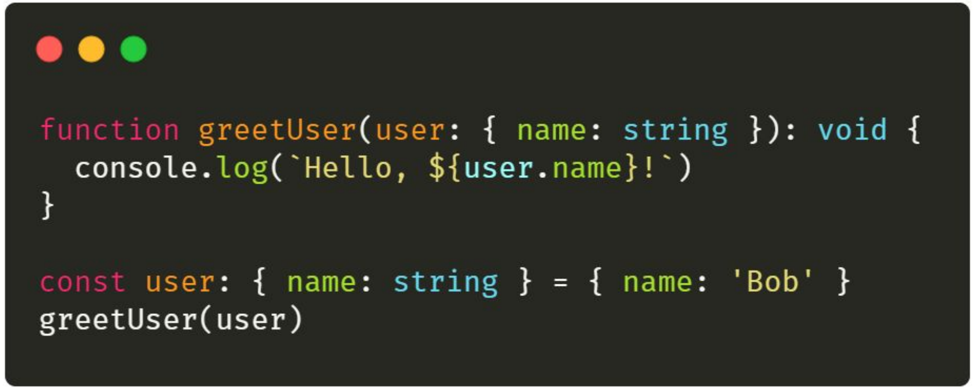
i

TIP: in your editor (e.g. VS Code), you can use ctrl + mouse cursor (or cmd + mouse cursor on Mac) to see the inferred type of an expression.

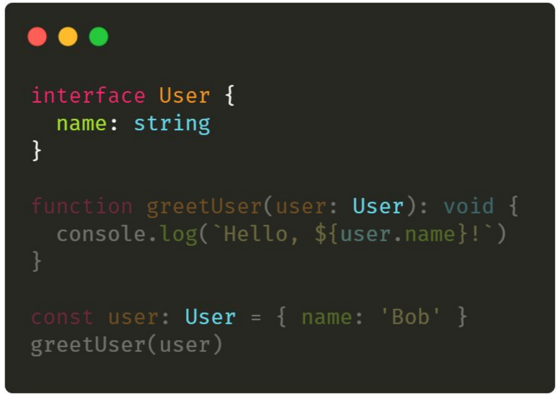
Interfaces

TypeScript type checking focuses on the **shape** of a value (also called “duck typing” or “structural subtyping”).

Interfaces can be used to **name** the types that describe the **shape** (or **structure**) of a set of values.



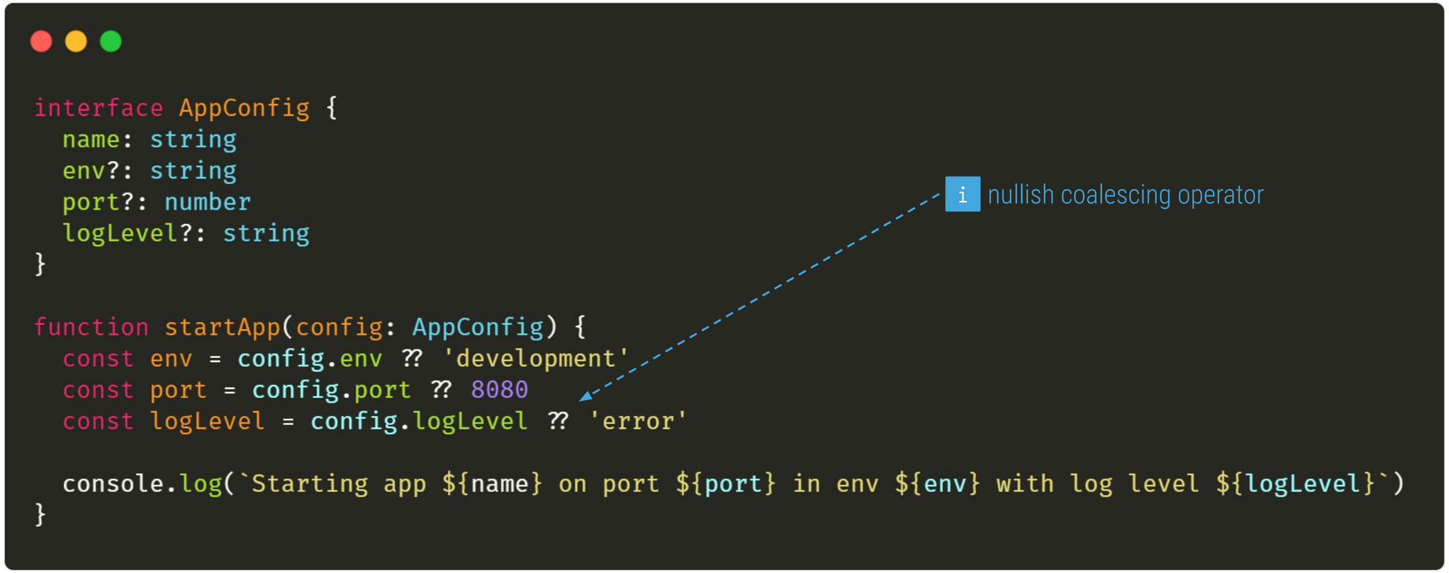
```
function greetUser(user: { name: string }): void {
  console.log(`Hello, ${user.name}!`)
}

const user: { name: string } = { name: 'Bob' }
greetUser(user)
```



```
interface User {
  name: string
}

function greetUser(user: User): void {
  console.log(`Hello, ${user.name}!`)
}

const user: User = { name: 'Bob' }
greetUser(user)
```

Interfaces

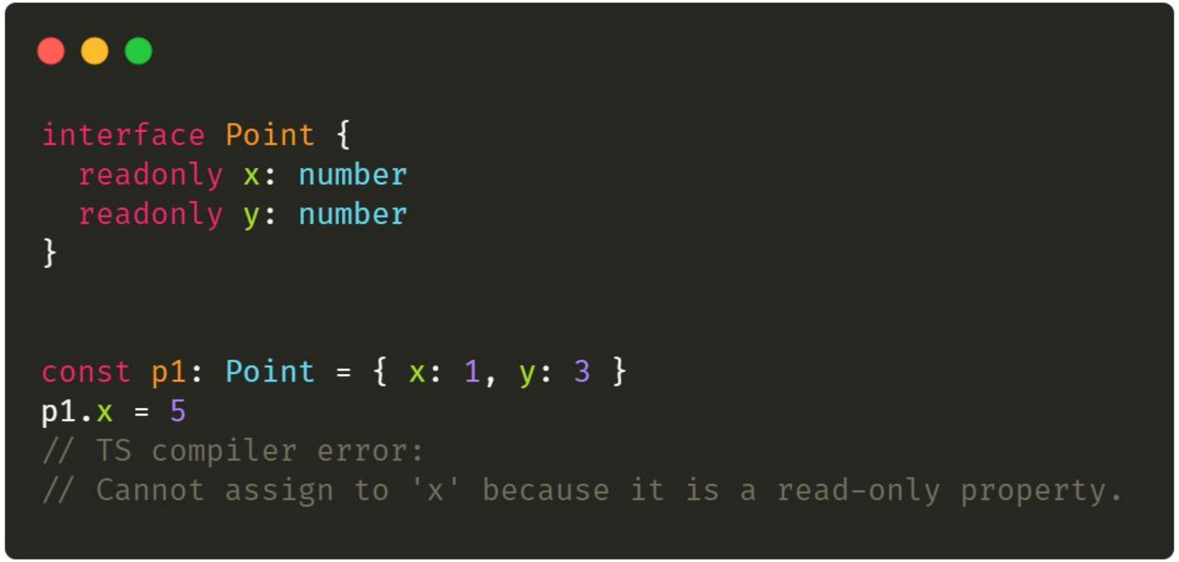
Some properties of an interface can be optional.



```
interface AppConfig {
  name: string
  env?: string
  port?: number
  logLevel?: string
}

function startApp(config: AppConfig) {
  const env = config.env ?? 'development'
  const port = config.port ?? 8080
  const logLevel = config.logLevel ?? 'error'

  console.log(`Starting app ${name} on port ${port} in env ${env} with log level ${logLevel}`)
}
```

Interfaces

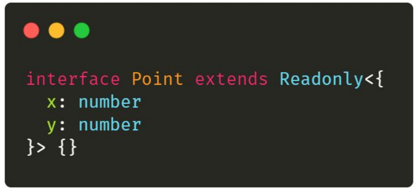
Some properties of an interface can be made “read only”.



```
interface Point {
  readonly x: number
  readonly y: number
}


const p1: Point = { x: 1, y: 3 }
p1.x = 5
// TS compiler error:
// Cannot assign to 'x' because it is a read-only property.
```

i

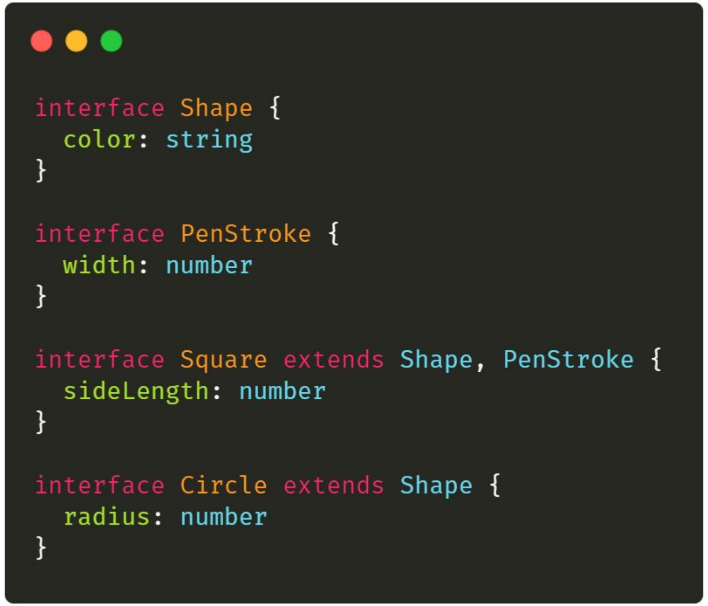
You can make a data type **immutable** at compile time (not at runtime though) by using the `readonly` modifier on each of its properties. You can also use the `Readonly<A>` mapped type to make all the properties of a type read-only.



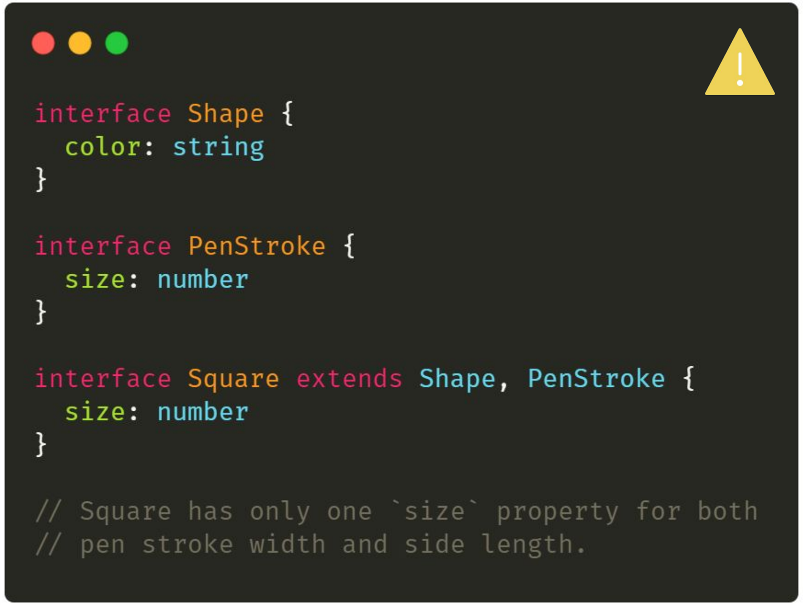
```
interface Point extends Readonly<{
  x: number
  y: number
}> {}
```

Interfaces

Interfaces can extend other interfaces, thus favoring reusable “components”.



```
interface Shape {
  color: string
}

interface PenStroke {
  width: number
}

interface Square extends Shape, PenStroke {
  sideLength: number
}

interface Circle extends Shape {
  radius: number
}
```



```
interface Shape {
  color: string
}

interface PenStroke {
  size: number
}

interface Square extends Shape, PenStroke {
  size: number
}

// Square has only one `size` property for both
// pen stroke width and side length.
```

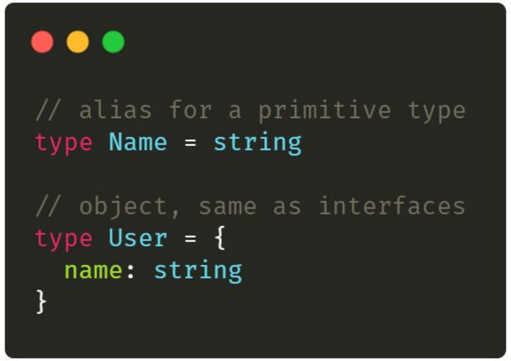
Type aliases

Type aliases can also be used to **name** types that describe the **shape** (or **structure**) of a set of values.

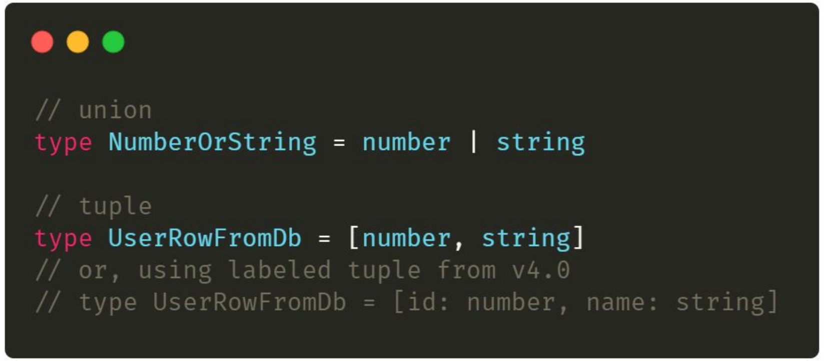
Unlike interfaces, they can be used for **primitive** types, **unions** and **tuples**.

i

Aliasing a primitive is not terribly useful, though it can be used as a form of documentation.



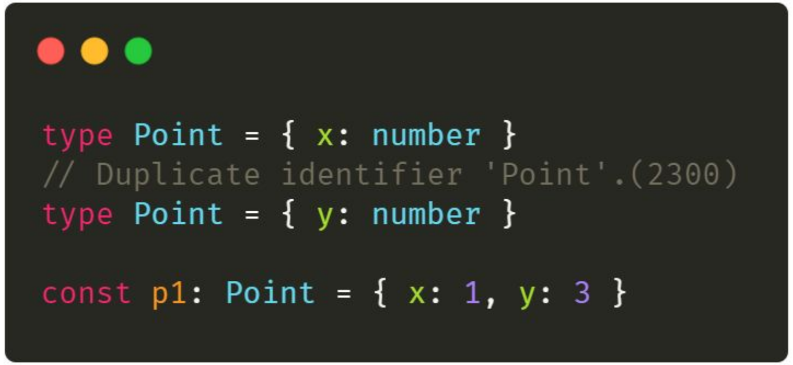
```
// alias for a primitive type
type Name = string

// object, same as interfaces
type User = {
  name: string
}
```



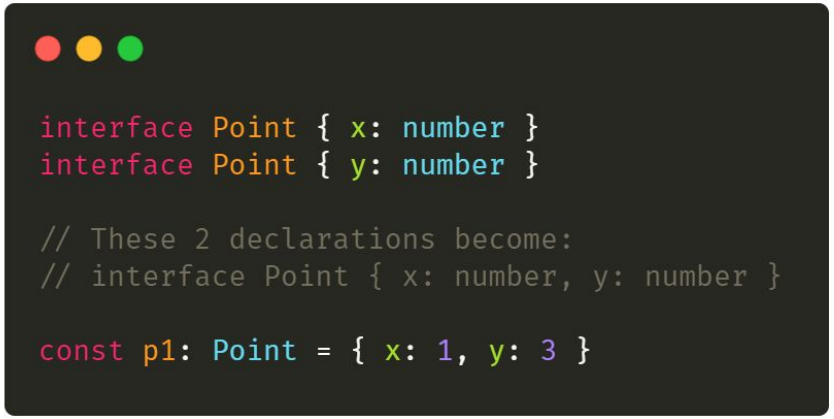
```
// union
type NumberOrString = number | string

// tuple
type UserRowFromDb = [number, string]
// or, using labeled tuple from v4.0
// type UserRowFromDb = [id: number, name: string]
```

Type aliases

Almost all features of an `interface` are available in `type`. However, a type alias **cannot be reopened** to add new properties, unlike an interface which is always extendable.



```
type Point = { x: number }
// Duplicate identifier 'Point'.(2300)
type Point = { y: number }

const p1: Point = { x: 1, y: 3 }
```

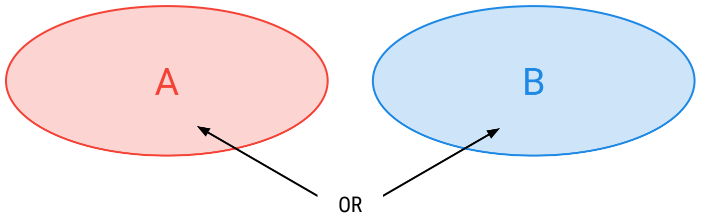


```
interface Point { x: number }
interface Point { y: number }

// These 2 declarations become:
// interface Point { x: number, y: number }

const p1: Point = { x: 1, y: 3 }
```

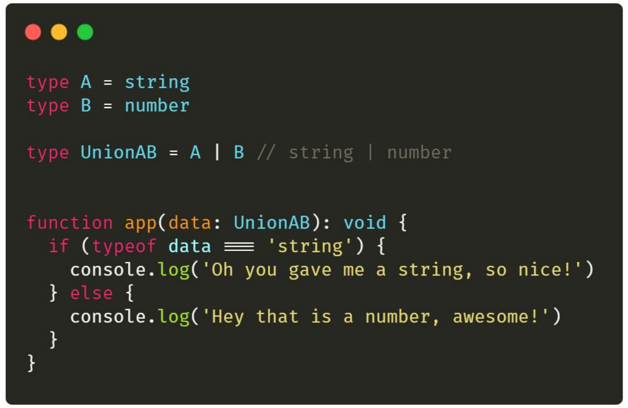
This is called "declaration merging".

Creating types from basic types

Union type



At a given time, a value is either of type A or of type B.

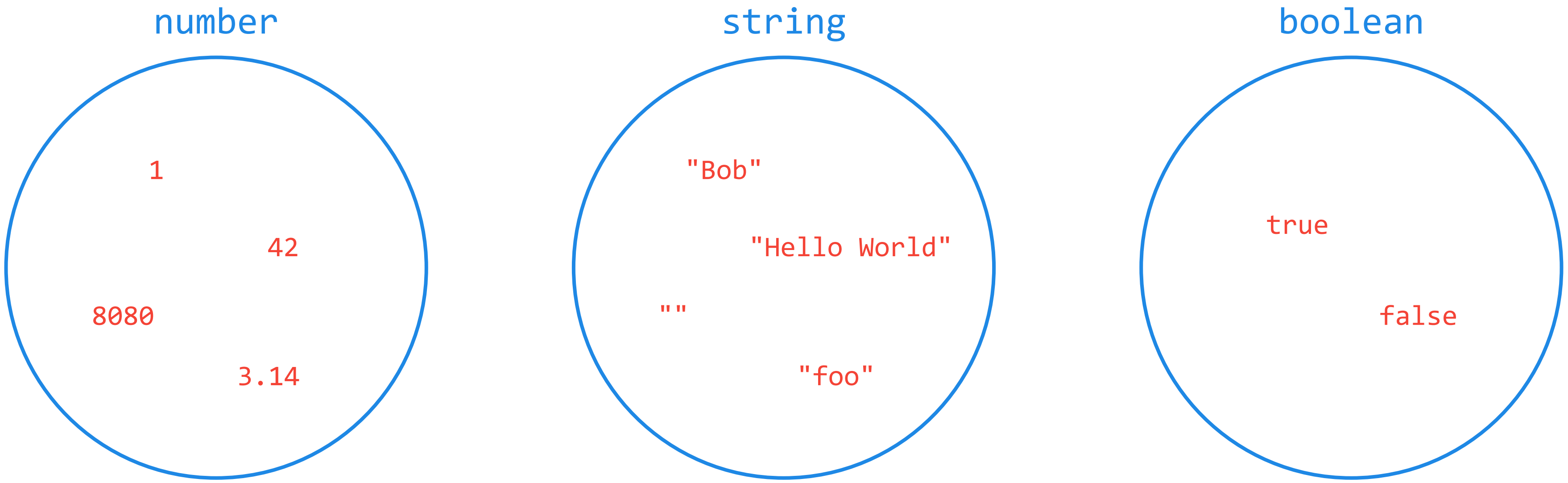


```
type A = string
type B = number

type UnionAB = A | B // string | number


function app(data: UnionAB): void {
  if (typeof data === 'string') {
    console.log('Oh you gave me a string, so nice!')
  } else {
    console.log('Hey that is a number, awesome!')
  }
}
```

Union types can be “named” with type aliases, but not with interfaces.

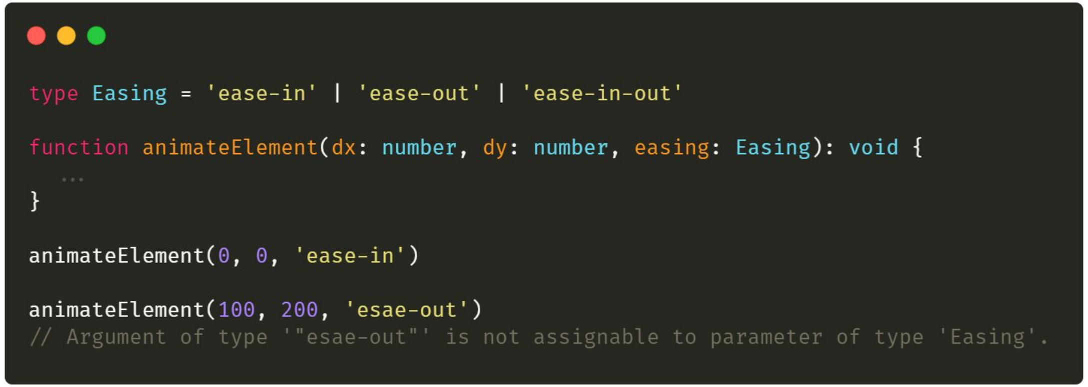
Literal types

A literal is a concrete sub-type of a collective type.

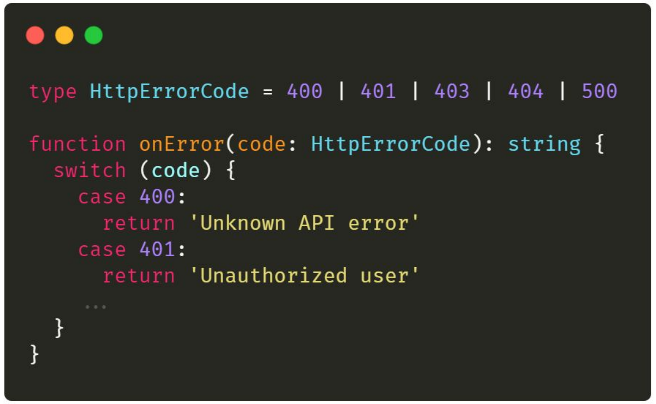


Collective types Concrete sub-types

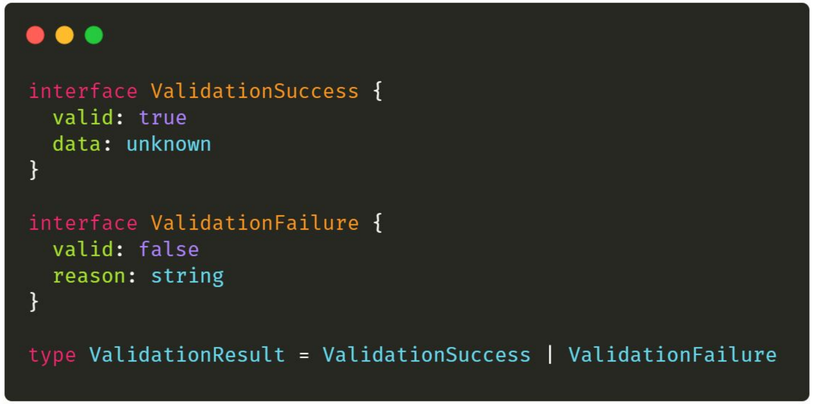
Literal types



```
type Easing = 'ease-in' | 'ease-out' | 'ease-in-out'

function animateElement(dx: number, dy: number, easing: Easing): void {
  ...
}

animateElement(0, 0, 'ease-in')

animateElement(100, 200, 'esae-out')
// Argument of type '"esae-out"' is not assignable to parameter of type 'Easing'.
```



```
type HttpErrorCode = 400 | 401 | 403 | 404 | 500

function onError(code: HttpErrorCode): string {
  switch (code) {
    case 400:
      return 'Unknown API error'
    case 401:
      return 'Unauthorized user'
    ...
  }
}
```



```
interface ValidationSuccess {
  valid: true
  data: unknown
}

interface ValidationFailure {
  valid: false
  reason: string
}

type ValidationResult = ValidationSuccess | ValidationFailure
```

Creating types from basic types

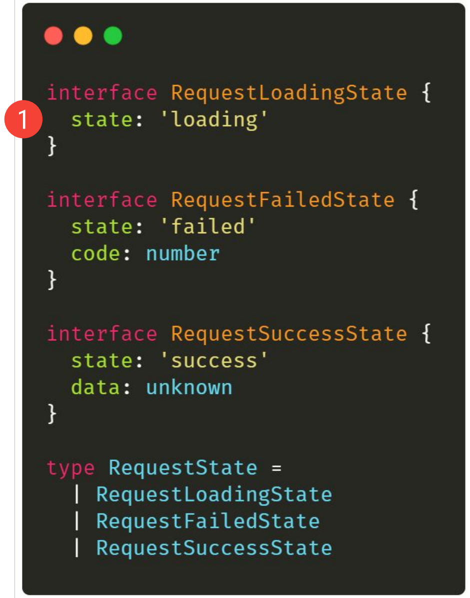
Discriminated union type

These types can be used to create **sum types** (functional programming concept).

1

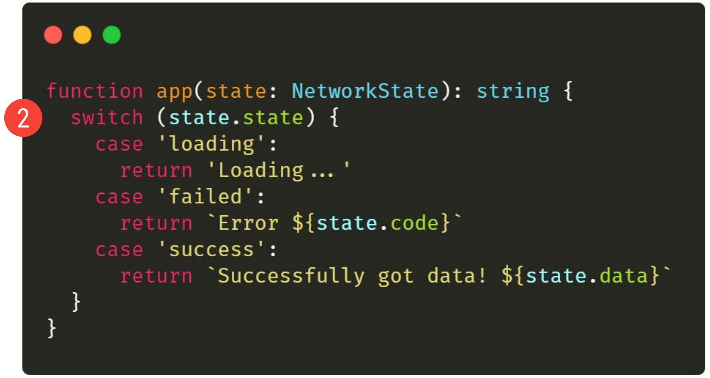
Union type with a **single field using a literal type** that lets TypeScript narrow down the possible “current” type.

2



1

```
interface RequestLoadingState {
  state: 'loading'
}

interface RequestFailedState {
  state: 'failed'
  code: number
}

interface RequestSuccessState {
  state: 'success'
  data: unknown
}

type RequestState =
  | RequestLoadingState
  | RequestFailedState
  | RequestSuccessState
```

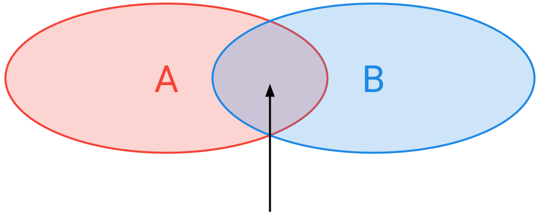


2

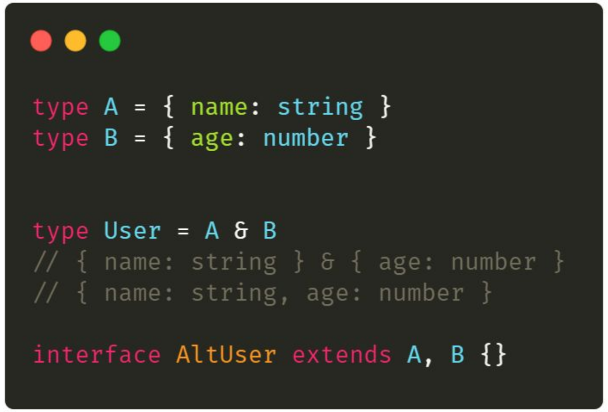
```
function app(state: NetworkState): string {
  switch (state.state) {
    case 'loading':
      return 'Loading... '
    case 'failed':
      return `Error ${state.code}`
    case 'success':
      return `Successfully got data! ${state.data}`
  }
}
```

Creating types from basic types

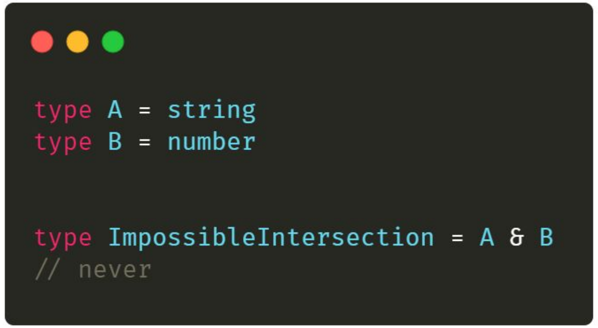
Intersection type



At a given time, a value has both the type A and the type B.



```
type A = { name: string }
type B = { age: number }

type User = A & B
// { name: string } & { age: number }
// { name: string, age: number }

interface AltUser extends A, B {}
```



```
type A = string
type B = number

type ImpossibleIntersection = A & B
// never
```

i

These types can be used to create **product types** (functional programming concept).

Parametric polymorphism (“generics”)

Polymorphism

It’s the provision of a **single interface** to work on **different types** of values.

There are 3 major classes of polymorphism:

- Ad hoc polymorphism
- Parametric polymorphism
- Subtype polymorphism

Parametric polymorphism (“generics”)

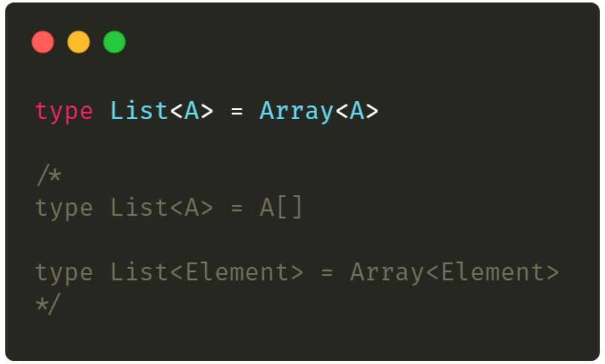
Parametric Polymorphism

A function or a data type can be written **generically** so that it can handle values identically, **no matter the differences between their types**.

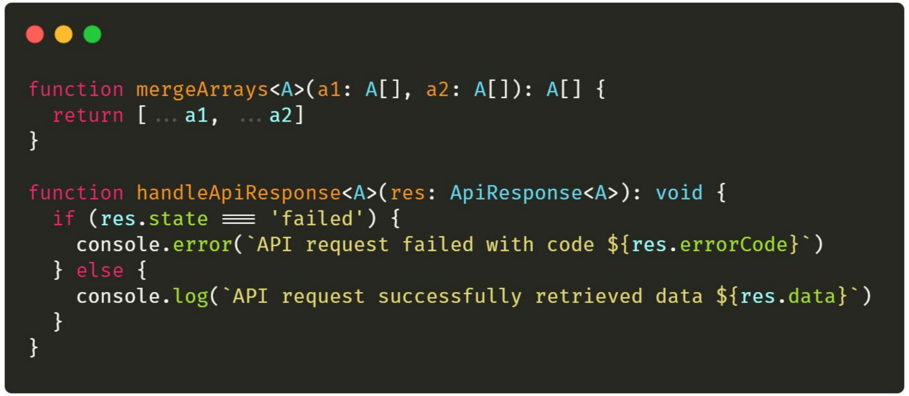
We’ve already crossed the path of a data type that uses parametric polymorphism...

Array!

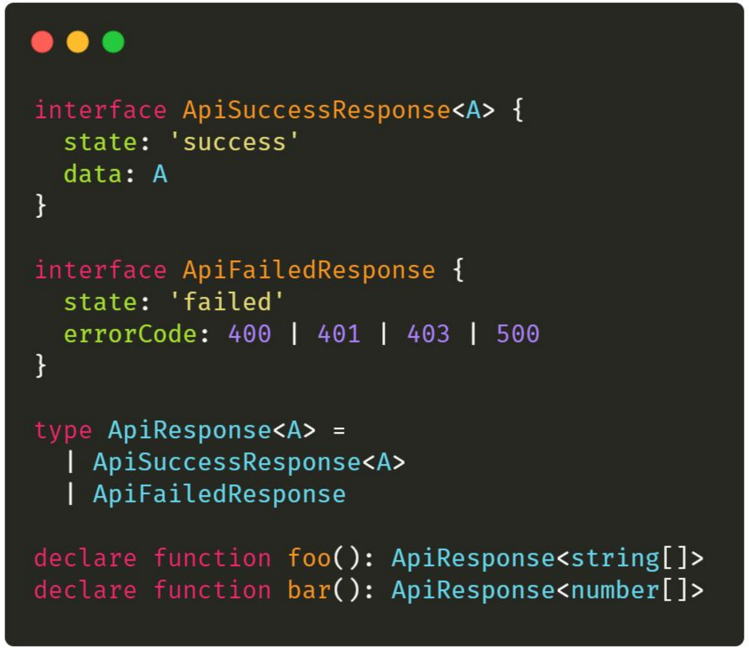
Parametric polymorphism (“generics”)



```typescript
type List<A> = Array<A>

/*
type List<A> = A[]

type List<Element> = Array<Element>
*/
```

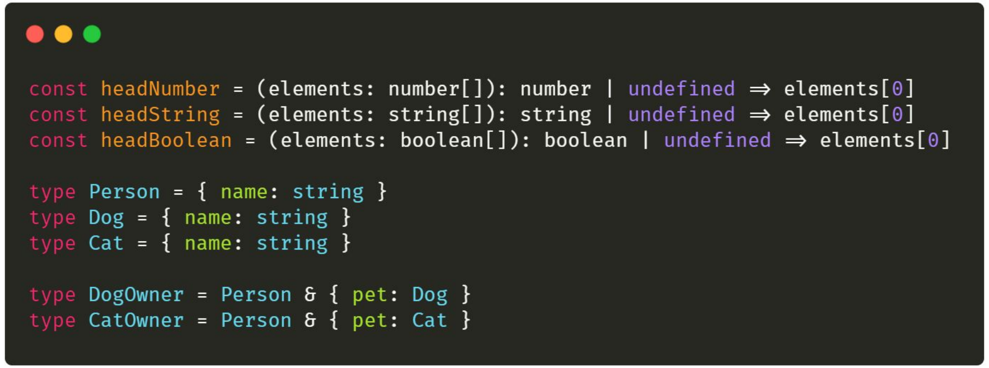


```typescript
function mergeArrays<A>(a1: A[], a2: A[]): A[] {
  return [...a1, ...a2]
}

function handleApiResponse<A>(res: ApiResponse<A>): void {
  if (res.state === 'failed') {
    console.error(`API request failed with code ${res.errorCode}`)
  } else {
    console.log(`API request successfully retrieved data ${res.data}`)
  }
}
```



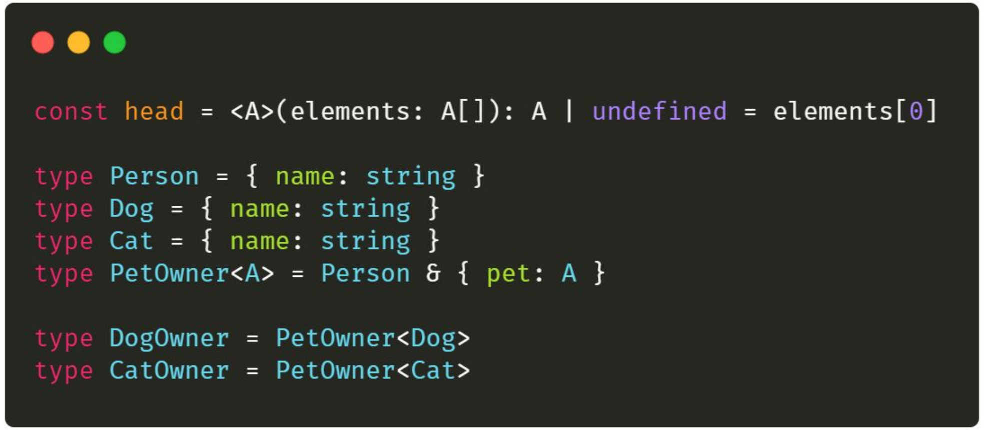
```typescript
interface ApiSuccessResponse<A> {
  state: 'success'
  data: A
}

interface ApiFailedResponse {
  state: 'failed'
  errorCode: 400 | 401 | 403 | 500
}

type ApiResponse<A> =
  | ApiSuccessResponse<A>
  | ApiFailedResponse

declare function foo(): ApiResponse<string[]>
declare function bar(): ApiResponse<number[]>
```

Parametric polymorphism (“generics”)



```
const headNumber = (elements: number[]): number | undefined => elements[0]
const headString = (elements: string[]): string | undefined => elements[0]
const headBoolean = (elements: boolean[]): boolean | undefined => elements[0]

type Person = { name: string }
type Dog = { name: string }
type Cat = { name: string }

type DogOwner = Person & { pet: Dog }
type CatOwner = Person & { pet: Cat }
```

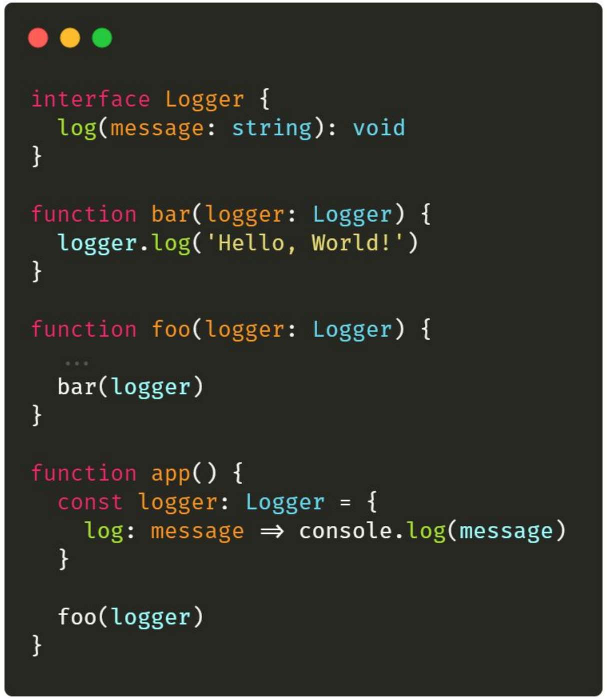
It helps avoid repetition.



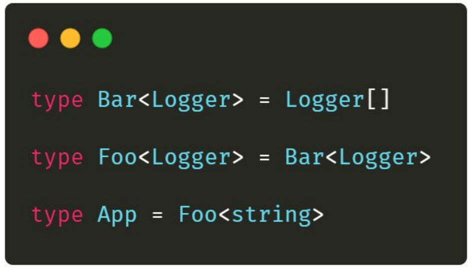
```
const head = <A>(elements: A[]): A | undefined = elements[0]

type Person = { name: string }
type Dog = { name: string }
type Cat = { name: string }
type PetOwner<A> = Person & { pet: A }

type DogOwner = PetOwner<Dog>
type CatOwner = PetOwner<Cat>
```

Parametric polymorphism (“generics”)

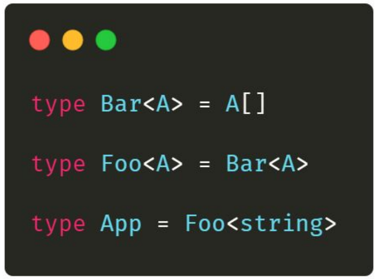
You can think of a data type with parameter(s) as a “function in the types world”.



```typescript
interface Logger {
  log(message: string): void
}

function bar(logger: Logger) {
  logger.log('Hello, World!')
}

function foo(logger: Logger) {
  ...
  bar(logger)
}

function app() {
  const logger: Logger = {
    log: message => console.log(message)
  }

  foo(logger)
}
```



```typescript
type Bar<Logger> = Logger[]

type Foo<Logger> = Bar<Logger>

type App = Foo<string>
```



```typescript
type Bar<A> = A[]

type Foo<A> = Bar<A>

type App = Foo<string>
```

How to use TypeScript

Installing TypeScript

Unless you are using Deno, you need to **install** TypeScript to use it in your project. The best way to do that is via npm:

```
npm init -y
```

This will create a package.json file at the root of the current directory.

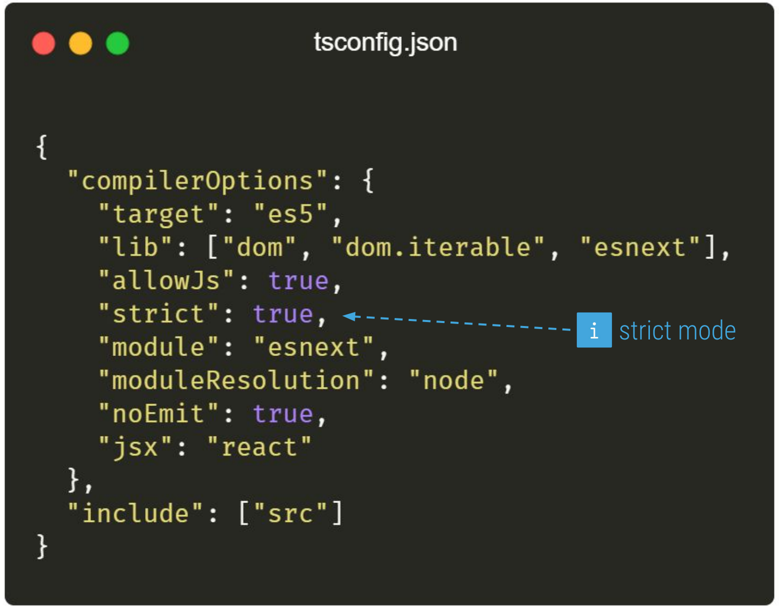
```
npm install --save typescript
```

This will install TypeScript in the node_modules directory.

```
tsc --init
```

This will create a tsconfig.json file at the root of the current directory.

Project configuration



```
{
  "compilerOptions": {
    "target": "es5",
    "lib": ["dom", "dom.iterable", "esnext"],
    "allowJs": true,
    "strict": true,
    "module": "esnext",
    "moduleResolution": "node",
    "noEmit": true,
    "jsx": "react"
  },
  "include": ["src"]
}
```

Example of a configuration file for a React project.

The presence of a **tsconfig.json** file in a directory indicates that the directory is the root of a TypeScript project.

This file specifies the root files and the **compiler options** required to compile the project.

(TSConfig options)

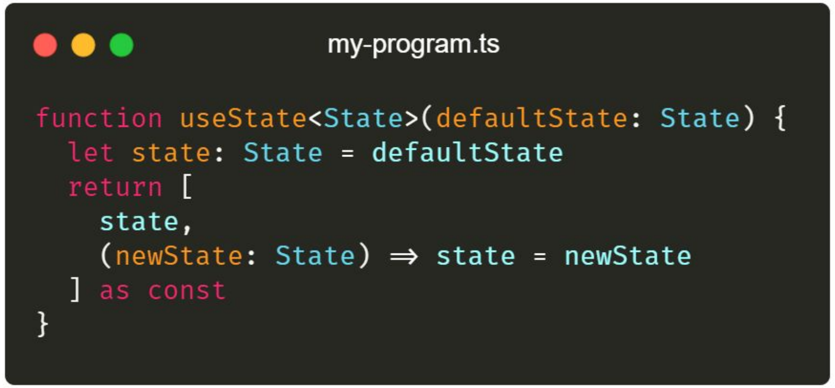
Declaration files

These files contain **type definitions only** (no runtime code).

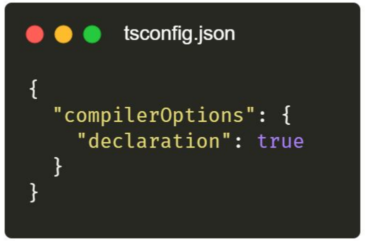
They are either:

- generated by **compiling a TS project** with the appropriate TS configuration,
- or written by people to **add types to existing JS projects**.

Declaration files

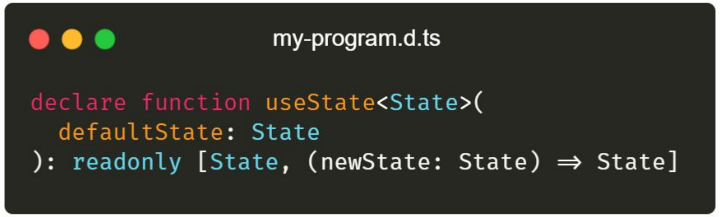

my-program.ts

```ts
function useState<State>(defaultState: State) {
  let state: State = defaultState
  return [
    state,
    (newState: State) ⇒ state = newState
  ] as const
}
```

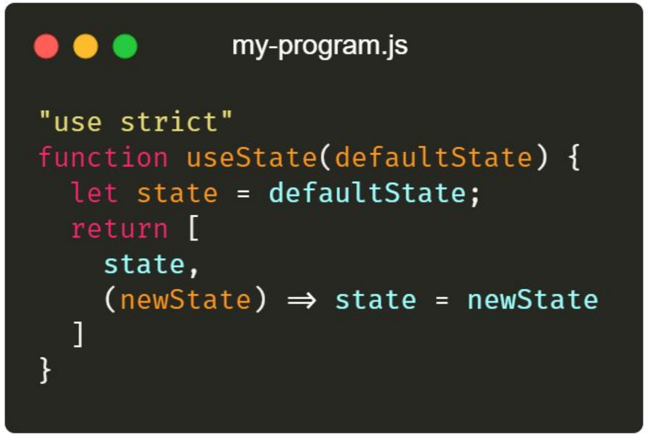

tsconfig.json

```json
{
  "compilerOptions": {
    "declaration": true
  }
}
```

Only type definitions, no runtime code.


my-program.d.ts

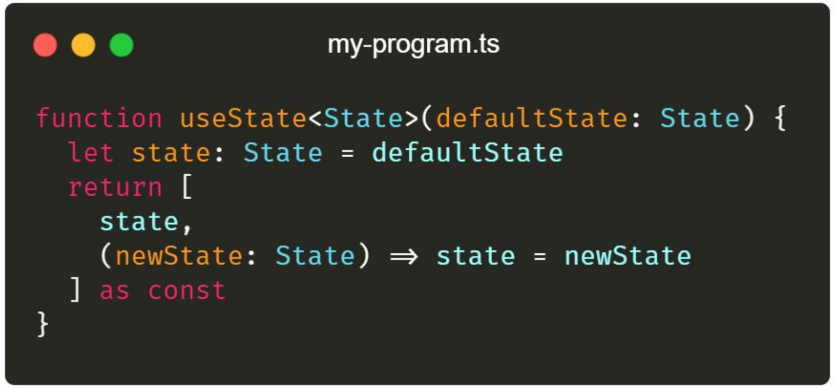
```ts
declare function useState<State>(
  defaultState: State
): readonly [State, (newState: State) ⇒ State]
```


my-program.js

```js
"use strict"
function useState(defaultState) {
  let state = defaultState;
  return [
    state,
    (newState) ⇒ state = newState
  ]
}
```

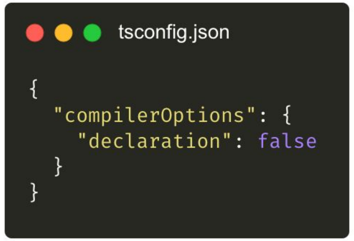
Runtime code (JavaScript).

Declaration files



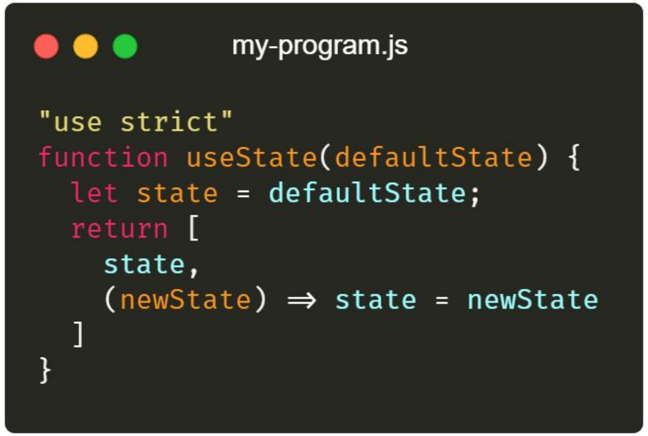
my-program.ts

```ts
function useState<State>(defaultState: State) {
  let state: State = defaultState
  return [
    state,
    (newState: State) => state = newState
  ] as const
}
```



tsconfig.json

```json
{
  "compilerOptions": {
    "declaration": false
  }
}
```



my-program.js

```js
"use strict"
function useState(defaultState) {
  let state = defaultState;
  return [
    state,
    (newState) => state = newState
  ]
}
```

Runtime code (JavaScript).

Declaration files

People can add types to JS libraries by contributing to the DefinitelyTyped GitHub project.

The **@types** organisation was added on the npm registry to get type definitions that are available in the DefinitelyTyped project.

More information is available in the TypeScript handbook.

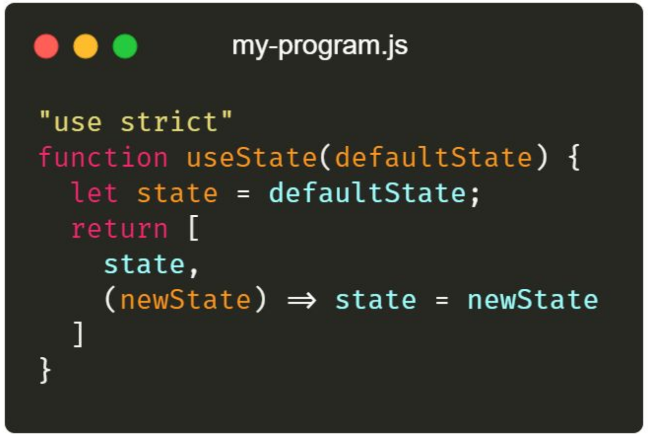
Declaration files

Example: You have installed React, Jest and Lodash libraries, but the types are missing? You can add them by using:

```
npm install --save @types/react @types/jest @types/lodash
```

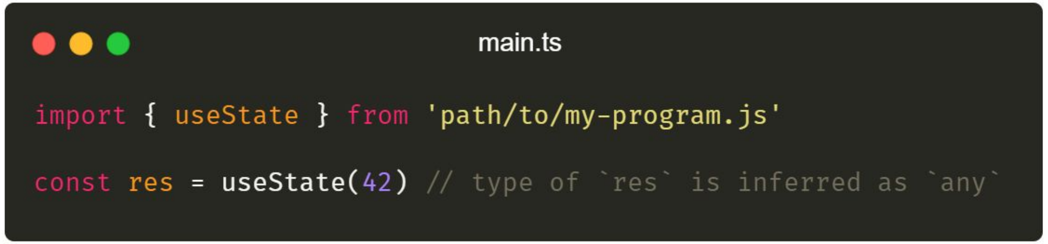
Sometimes, the types are already available in the original packages and you don’t have to manually install them.

Declaration files



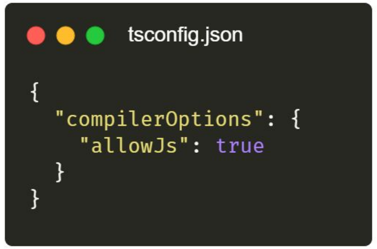
my-program.js

```js
"use strict"
function useState(defaultState) {
  let state = defaultState;
  return [
    state,
    (newState) => state = newState
  ]
}
```



main.ts

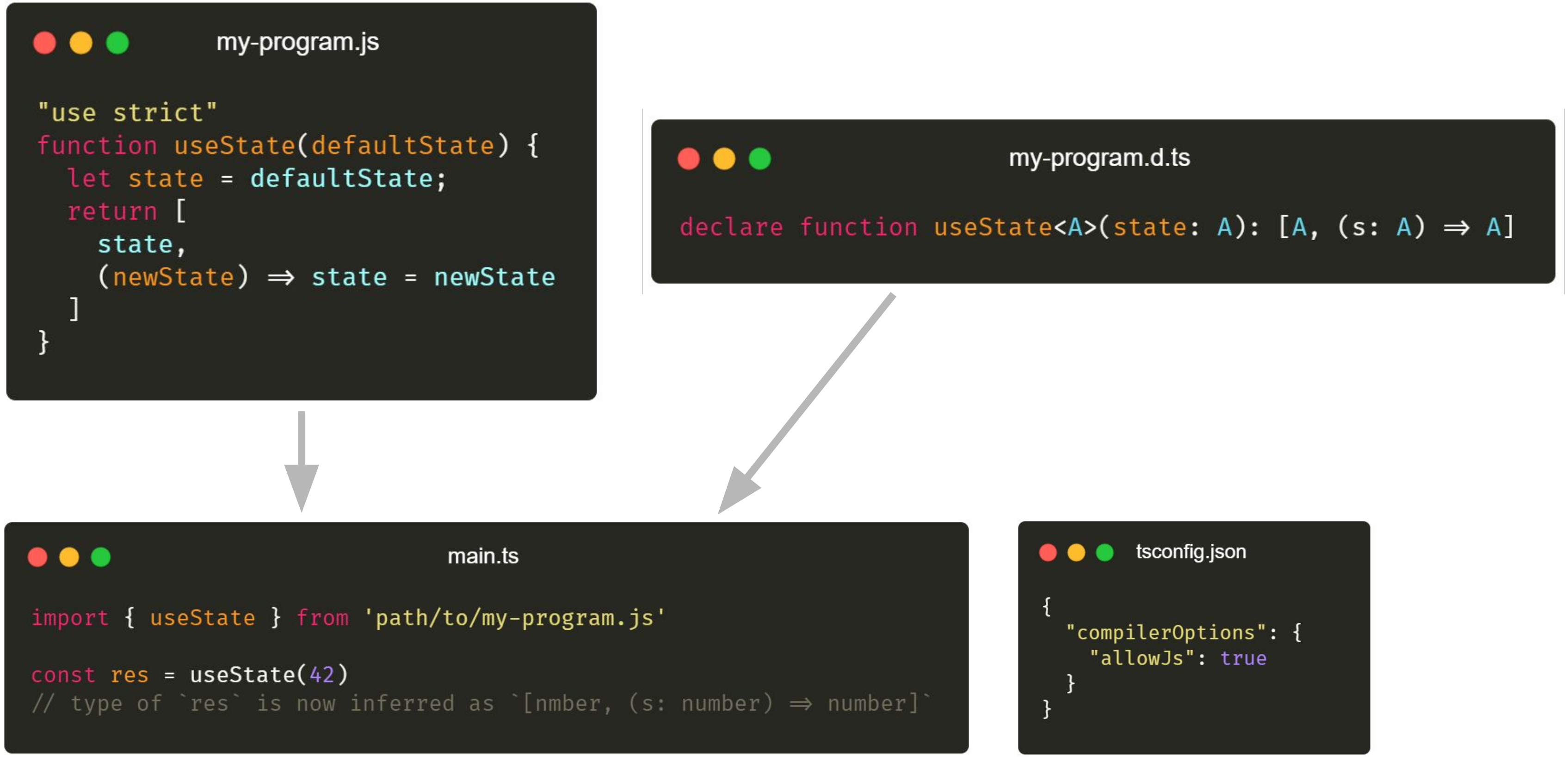
```ts
import { useState } from 'path/to/my-program.js'

const res = useState(42) // type of `res` is inferred as `any`
```



tsconfig.json

```json
{
  "compilerOptions": {
    "allowJs": true
  }
}
```

Declaration files



my-program.js

```js
"use strict"
function useState(defaultState) {
  let state = defaultState;
  return [
    state,
    (newState) => state = newState
  ]
}
```

my-program.d.ts

```ts
declare function useState<A>(state: A): [A, (s: A) => A]
```

main.ts

```ts
import { useState } from 'path/to/my-program.js'

const res = useState(42)
// type of `res` is now inferred as `[nmber, (s: number) => number]`
```

tsconfig.json

```json
{
  "compilerOptions": {
    "allowJs": true
  }
}
```

Using TSC

The **tsc** command is available after installing TypeScript. It can be used to compile a TypeScript project.

```
npx tsc
```

```
npx tsc --project path/to/tsconfig-directory
```

```
npx tsc --outDir build
```

Using Parcel

Parcel is a web application bundler that requires zero configuration to work. It competes with tools such as Webpack and Rollup.

```
npm install -g parcel-bundler
```

There is a dedicated section in the Parcel documentation to use it with TypeScript.

Using Create React App

Create React App is a toolchain (bundling, linting, formatting...) to bootstrap React apps.

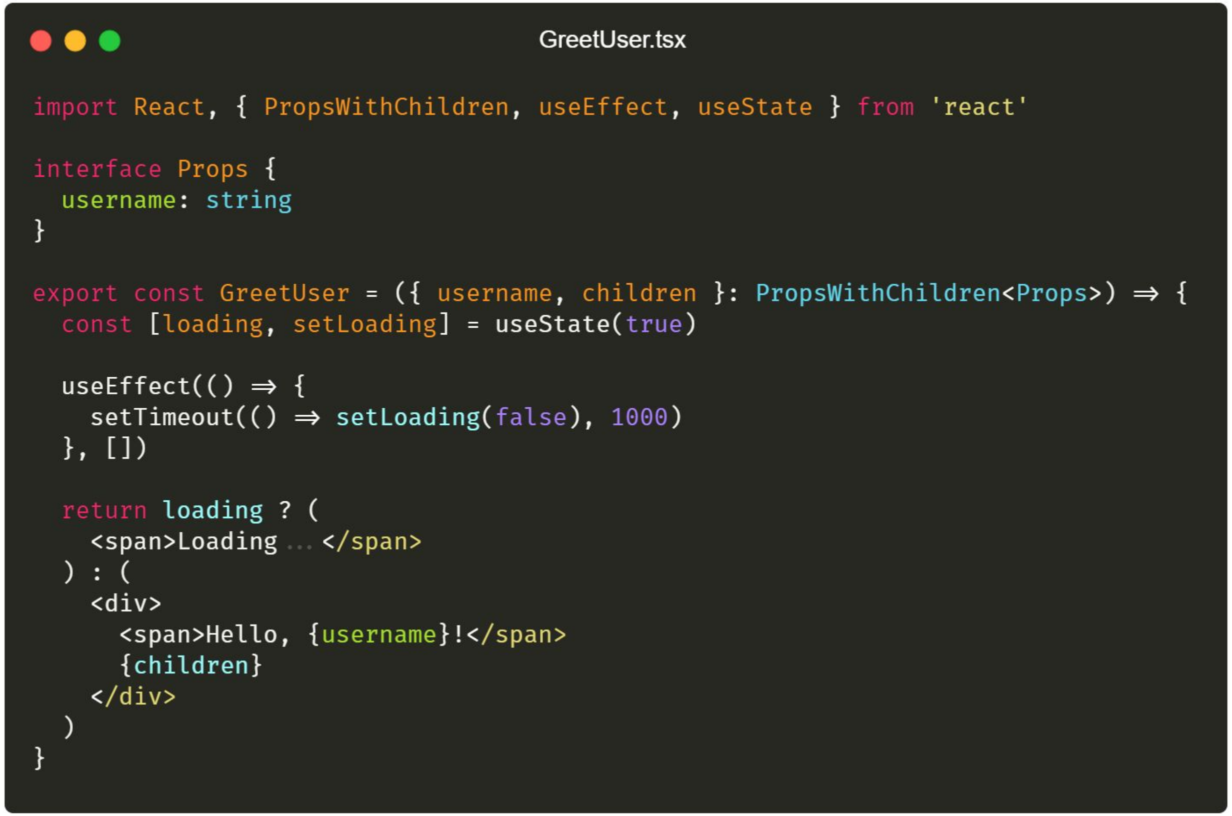
```
npm install -g create-react-app
```

This will install CRA globally.

```
npx create-react-app my-app --template typescript
```

This will create a my-app directory, then bootstrap a React app using TypeScript inside of it.

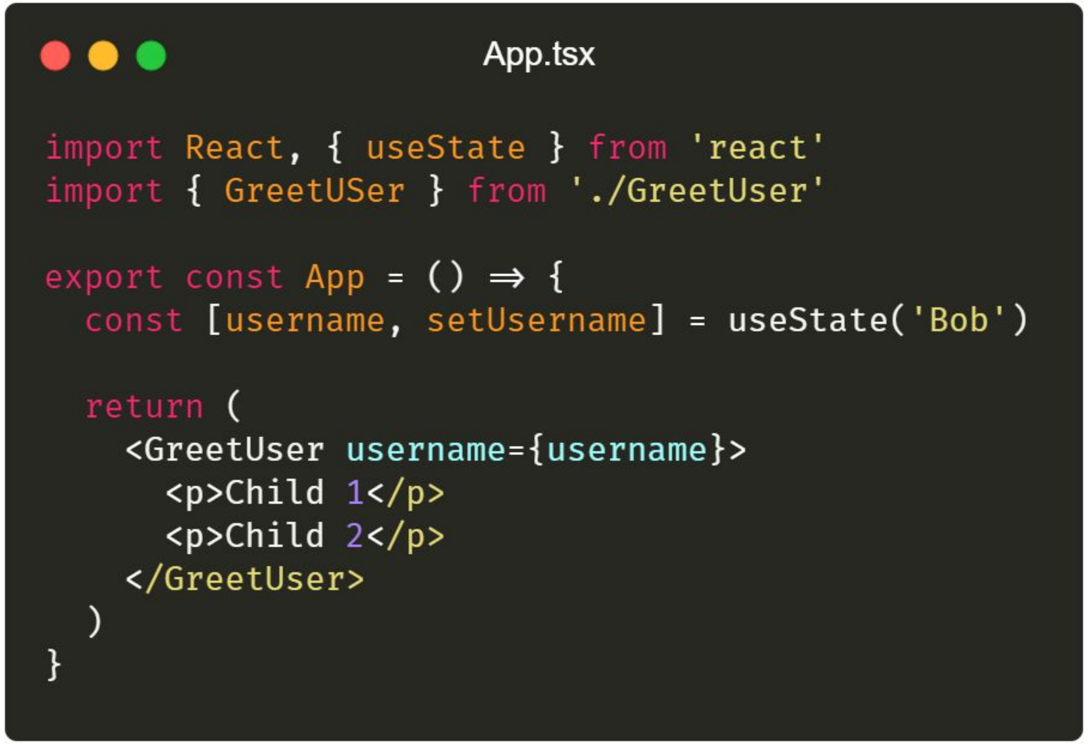
React functional component example



GreetUser.tsx

```tsx
import React, { PropsWithChildren, useEffect, useState } from 'react'

interface Props {
  username: string
}

export const GreetUser = ({ username, children }: PropsWithChildren<Props>) ⇒ {
  const [loading, setLoading] = useState(true)

  useEffect(() ⇒ {
    setTimeout(() ⇒ setLoading(false), 1000)
  }, [])

  return loading ? (
    <span>Loading...</span>
  ) : (
    <div>
      <span>Hello, {username}!</span>
      {children}
    </div>
  )
}
```

React functional component example


App.tsx

```
import React, { useState } from 'react'
import { GreetUSer } from './GreetUser'

export const App = () ⇒ {
  const [username, setUsername] = useState('Bob')

  return (
    <GreetUser username={username}>
      <p>Child 1</p>
      <p>Child 2</p>
    </GreetUser>
  )
}
```

Useful resources

- TypeScript playground
- The TypeScript handbook
- A glossary of TypeScript
- Type or treat challenges
- type-challenges repository
- TypeScript exercises