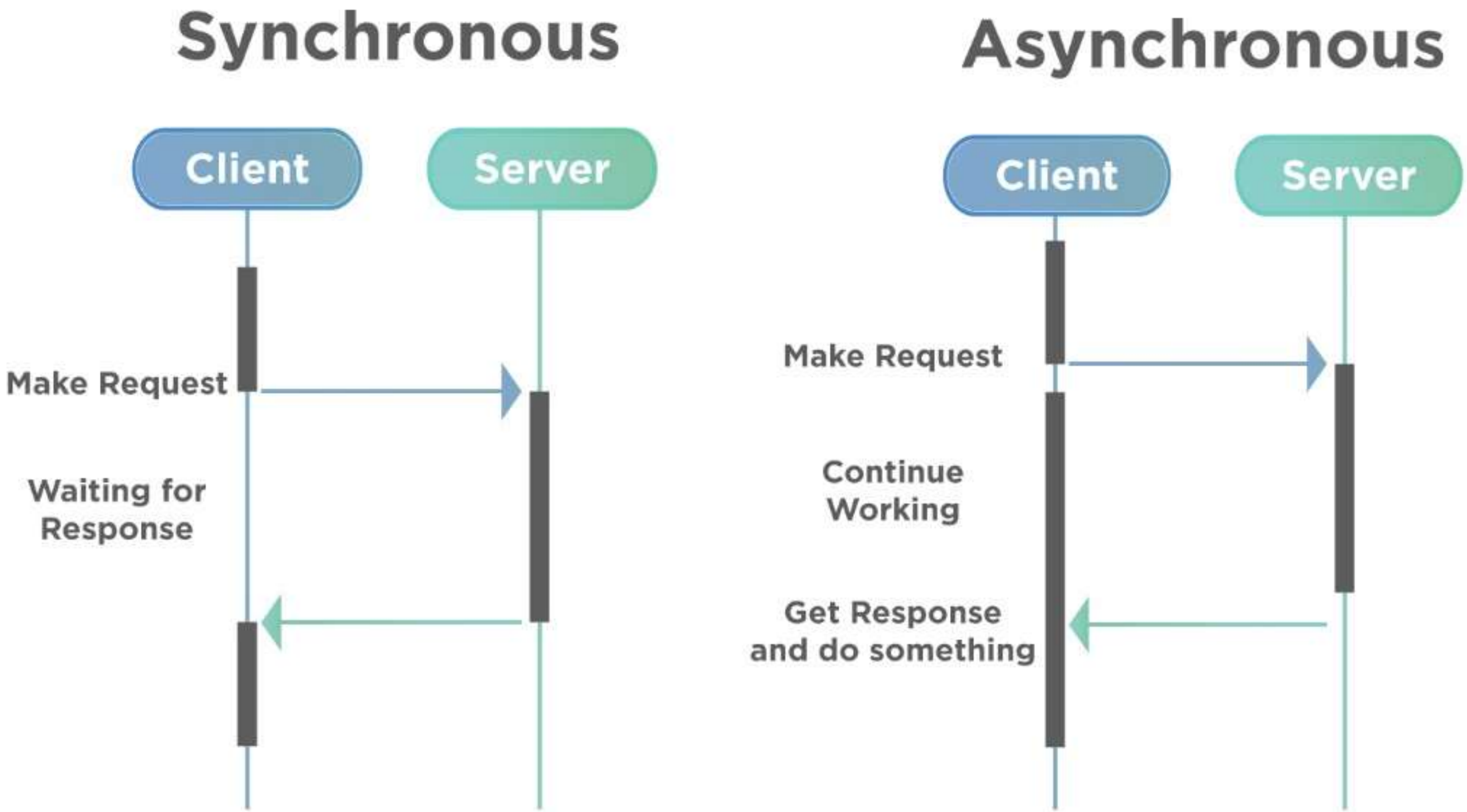
Asynchronous Programming in JS

Testleaf
Always Ahead

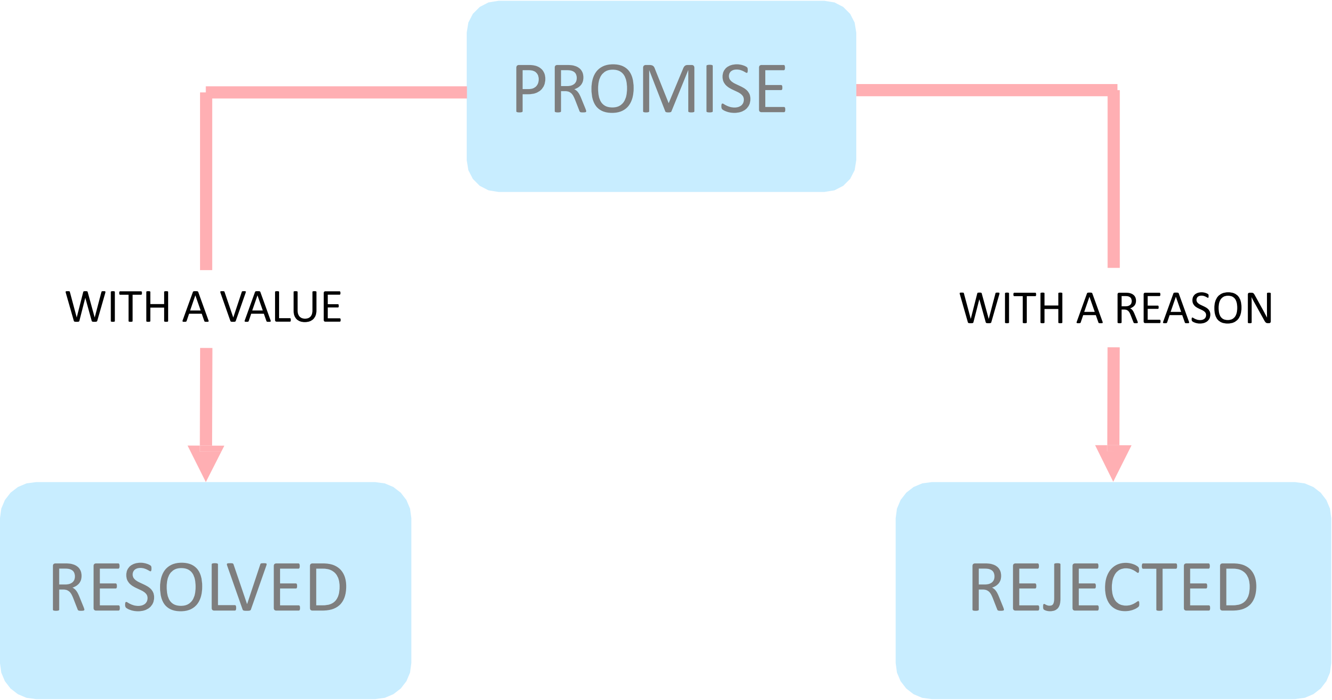
Agenda - 1

- Introduction to Asynchronous Programming
- Callbacks in JS
- Promises in JS
- Call back hell
- Async/Await

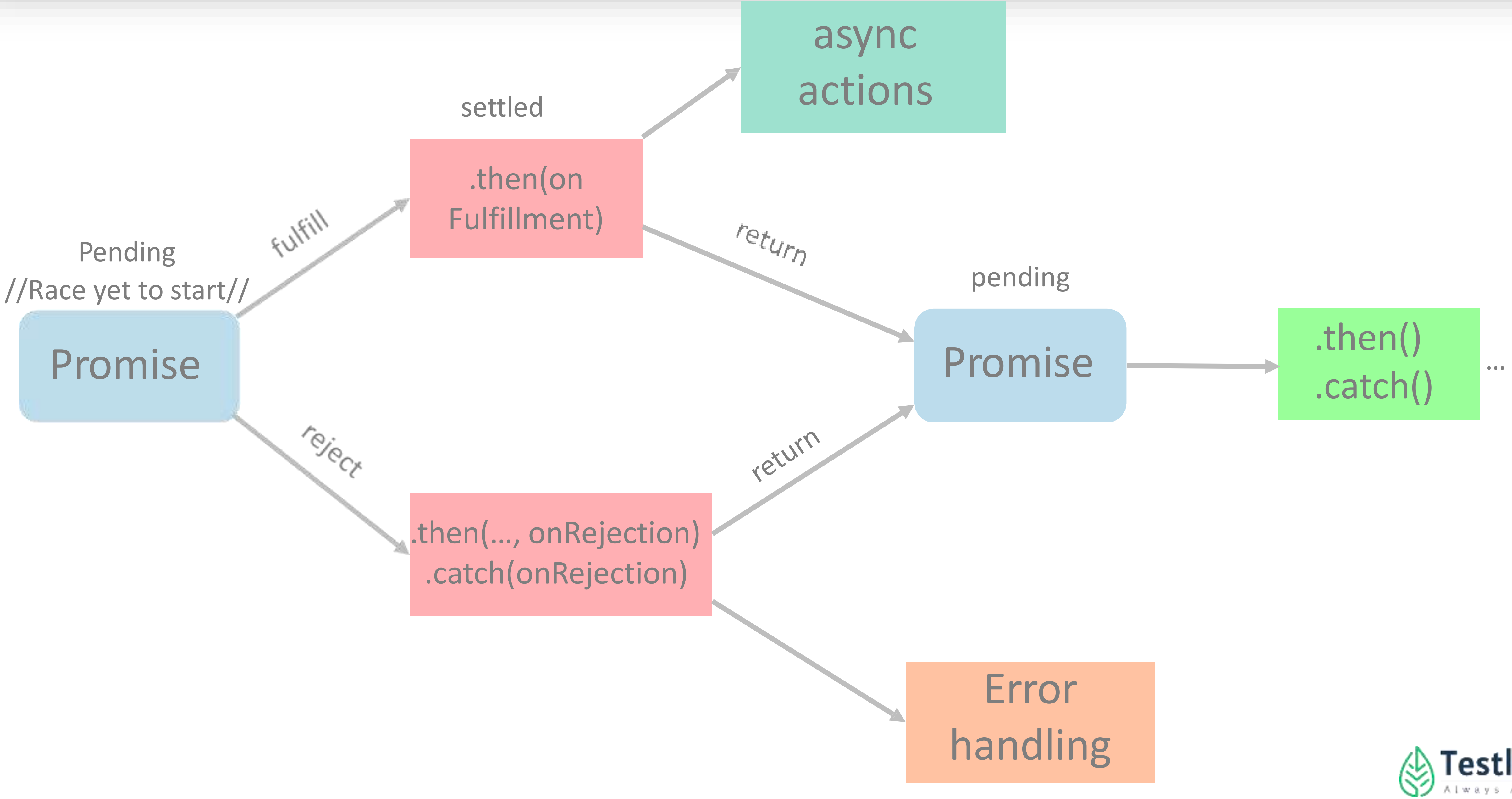
Synchronous vs Asynchronous



Promise in JS



Promise Flow



Testleaf Always Ahead

Promise Flow – With Relay Race analogy

```
let batonDelivery = new Promise((resolve, reject) => {

    let isBatonDelivered = true; // Simulate whether the baton handoff is successful

    if(isBatonDelivered) {

        resolve("Baton successfully passed! Keep running!");

    } else {

        reject("The baton was dropped. Race over.");

    }
});

// Using the promise
batonDelivery
    .then(message => {
        console.log(message);   // This runs if the baton was passed successfully
    })
    .catch(error => {
        console.log(error);     // This runs if the baton handoff failed
    });
```

Testleaf
Always Ahead

What and How Did Callback Hell Came About?

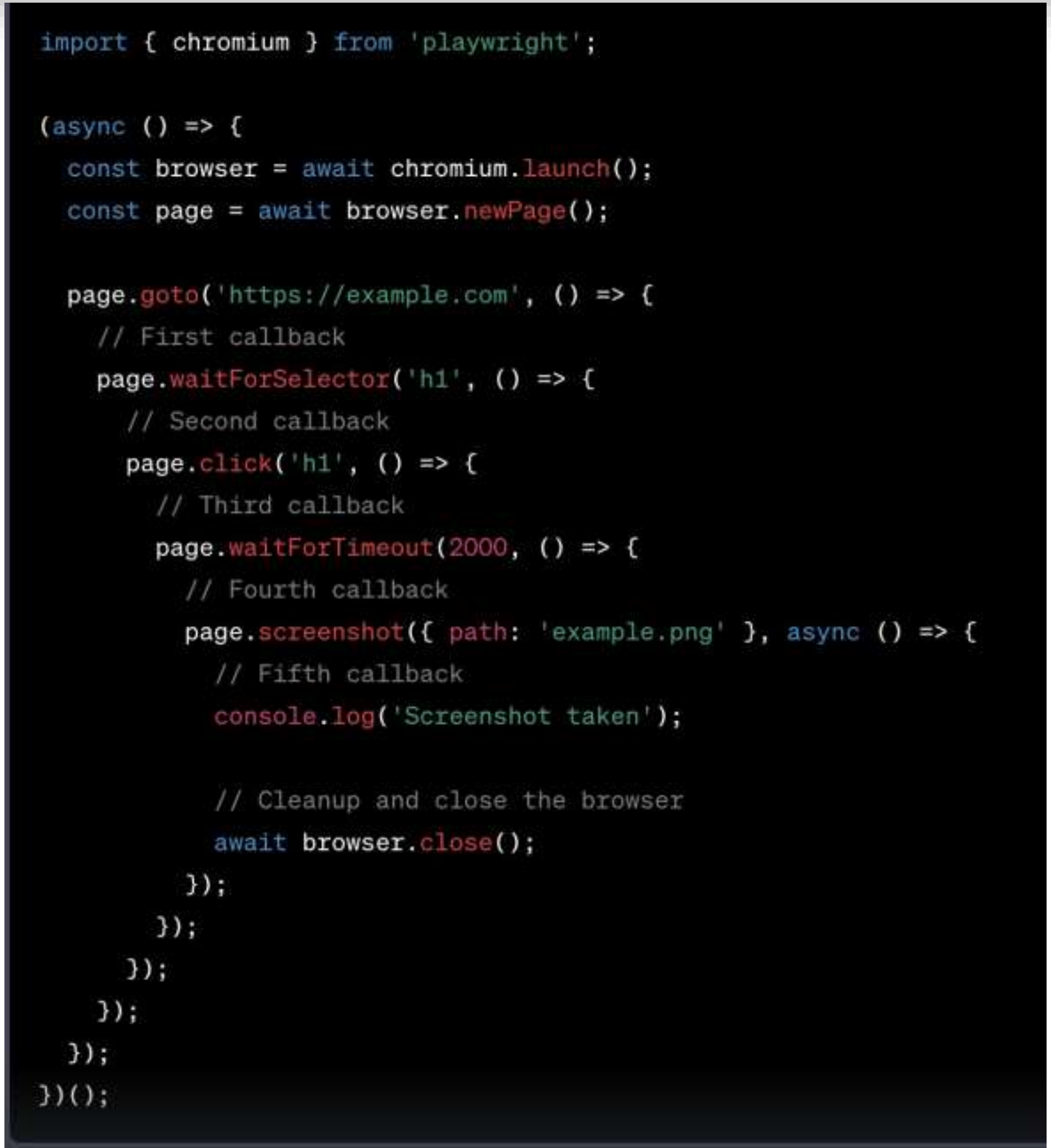
- Callback hell (also known as "Pyramid of Doom") emerged due to the way asynchronous JavaScript was originally handled, especially in early web development.

- Scenario where JavaScript is single-threaded, meaning it can only execute one operation at a time.

- However, web applications often need to:
  - ✓ Fetch data from a server
  - ✓ Read files
  - ✓ Wait for user input
  - ✓ Interact with databases.

  To avoid blocking the execution of code while waiting for these operations to complete, JavaScript used callbacks—functions that run once an asynchronous task is finished.

Call back hell



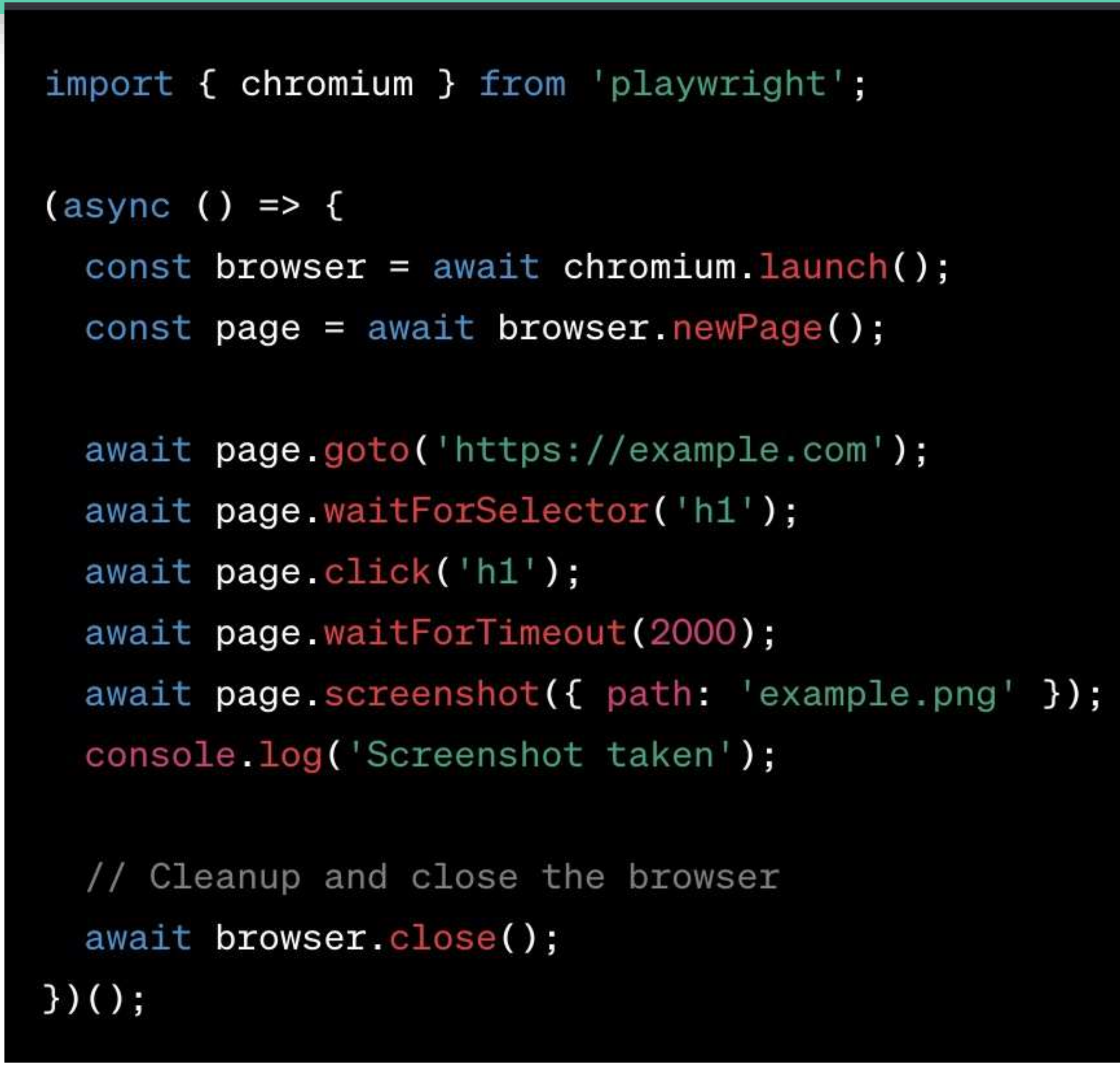
```
import { chromium } from 'playwright';

(async () => {
  const browser = await chromium.launch();
  const page = await browser.newPage();

  page.goto('https://example.com', () => {
    // First callback
    page.waitForSelector('h1', () => {
      // Second callback
      page.click('h1', () => {
        // Third callback
        page.waitForTimeout(2000, () => {
          // Fourth callback
          page.screenshot({ path: 'example.png' }, async () => {
            // Fifth callback
            console.log('Screenshot taken');

            // Cleanup and close the browser
            await browser.close();
          });
        });
      });
    });
  });
})();
```

Async/Await

To solve callback hell, JavaScript evolved in two major steps:

❑ **Promises (ES6, 2015)** – Introduced to flatten nested callbacks using .then()

❑ **Async/Await (ES8, 2017)** – Made asynchronous code look like synchronous code, making it much easier to read.



```
import { chromium } from 'playwright';

(async () => {
  const browser = await chromium.launch();
  const page = await browser.newPage();

  await page.goto('https://example.com');
  await page.waitForSelector('h1');
  await page.click('h1');
  await page.waitForTimeout(2000);
  await page.screenshot({ path: 'example.png' });
  console.log('Screenshot taken');

  // Cleanup and close the browser
  await browser.close();
})();
```

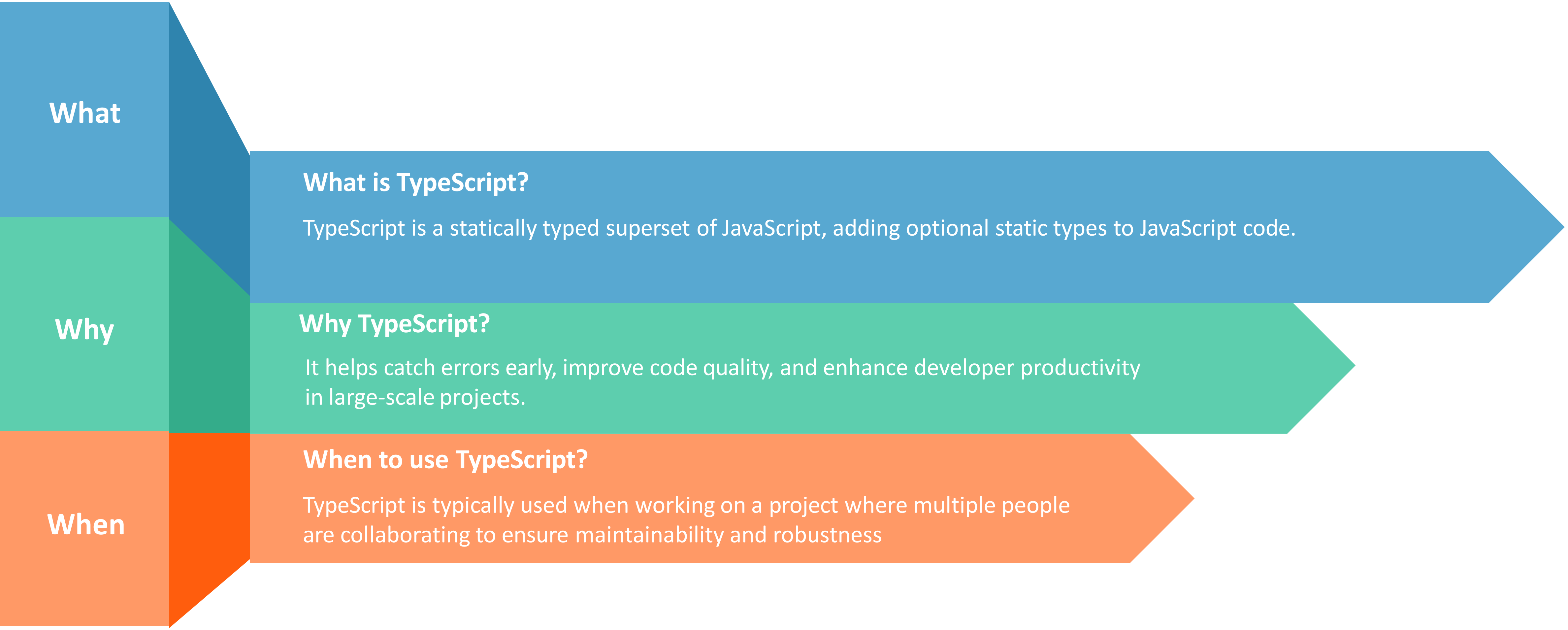
Agenda – Part 2 (Basics)

- Introduction to TypeScript
- Why TS?
- How to add a new TS project?
- Compile and Run

Let’s Learn TS Fundamentals

What, Why & When

The Golden Circle



What is TypeScript?

TypeScript is a statically typed superset of JavaScript, adding optional static types to JavaScript code.

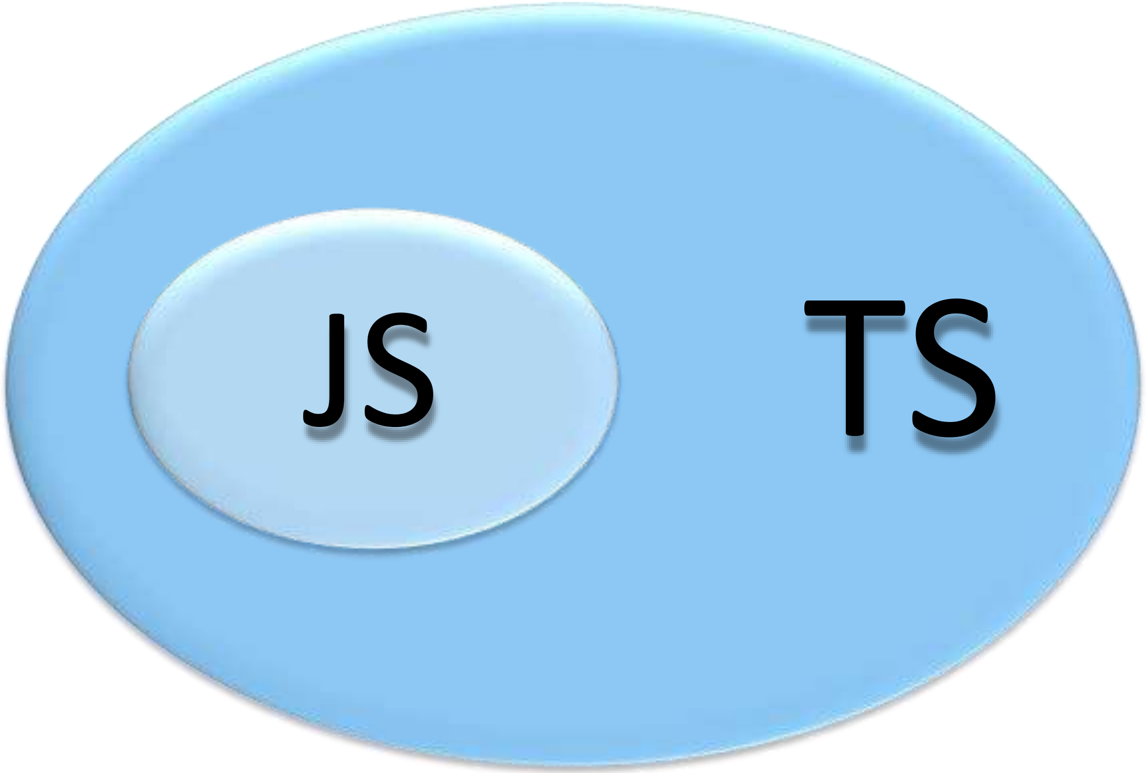
Why TypeScript?

It helps catch errors early, improve code quality, and enhance developer productivity in large-scale projects.

When to use TypeScript?

TypeScript is typically used when working on a project where multiple people are collaborating to ensure maintainability and robustness

Introduction to TypeScript



Why prefer TS?

PROS

- ✓ Has everything JavaScript has, plus additional features
- ✓ Supports static typing
- ✓ Discover errors at compile-time

CONS

- ✗ Takes longer to compile code

Testleaf
Always Ahead

Let's add a new TypeScript project

- ***Step 1:*** Create a new project
- ***Step 2****:* Open that project in your VS Code (IDE)
- ***Step 3****:* npm install typescript
- ***Step 4****:* tsc --version
- ***Step 5****:* tsc --init

Let’s compile and run

- Create a new file demo.ts on your IDE
- Add any JS code (e.g. console.log(“hi, i am TS”))
- Now run tsc demo.ts
- Observe if the demo.js is created
- Now run node demo.js

Agenda – 3 (Deep dive into Types)

- Built in types in TS
- Implicit vs Explicit Type Declaration
- Adding custom types
- Combining types

Testleaf
Always Ahead

Built in types in TS

JAVASCRIPT

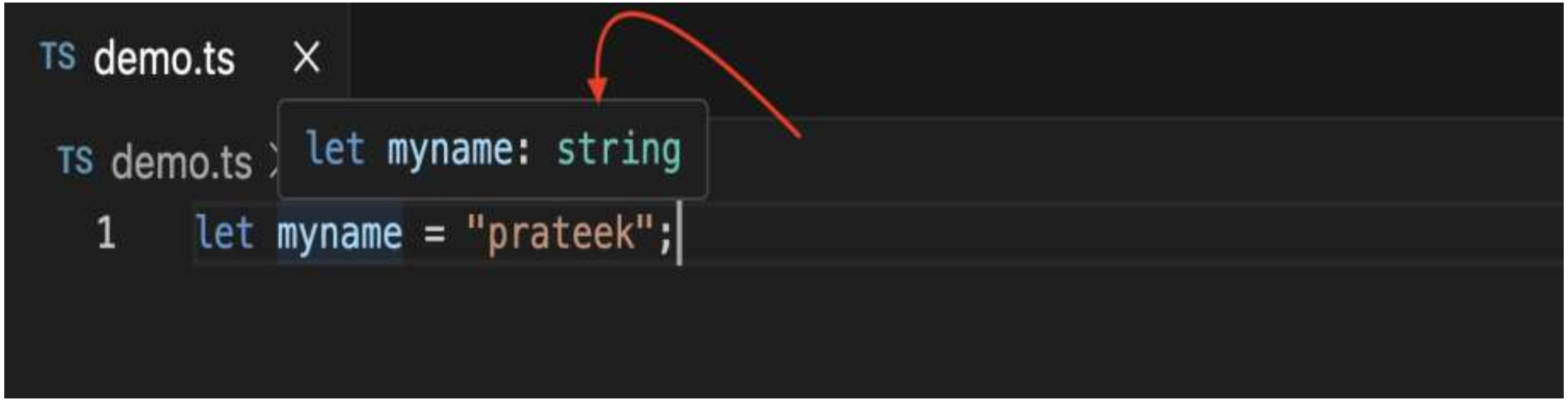
- number
- String
- boolean
- null
- undefined
- object

TYPESCRIPT

- any
- never
- enum
- tuple

Implicit Types vs Explicit Types

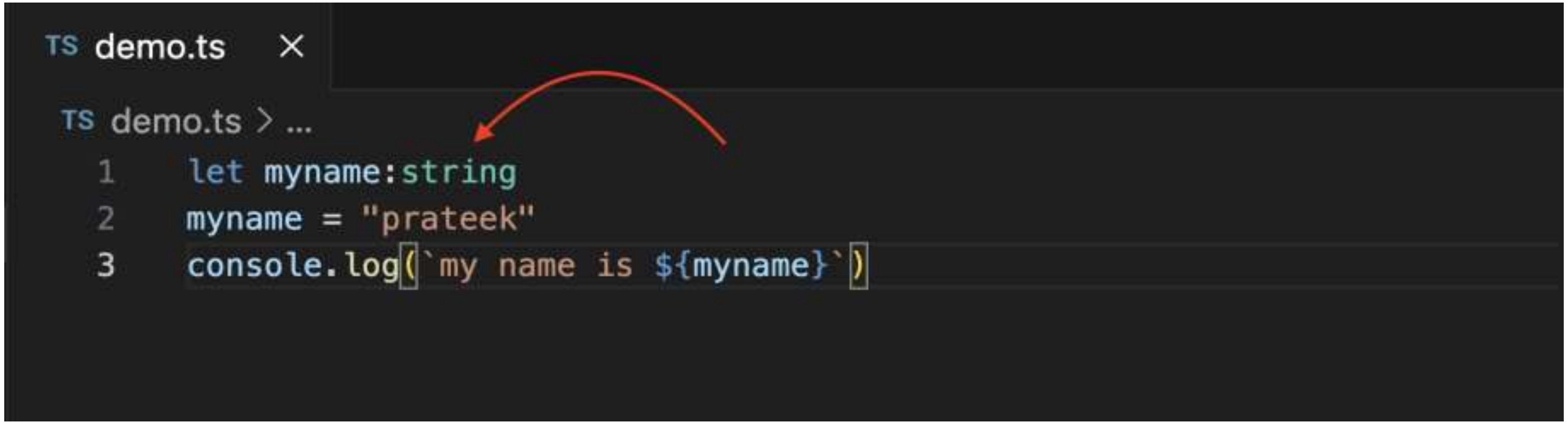
Implicit Types : means that the type is inferred by TypeScript type inference system which takes responsibility away from us of writing the types



```
let myname: string
```

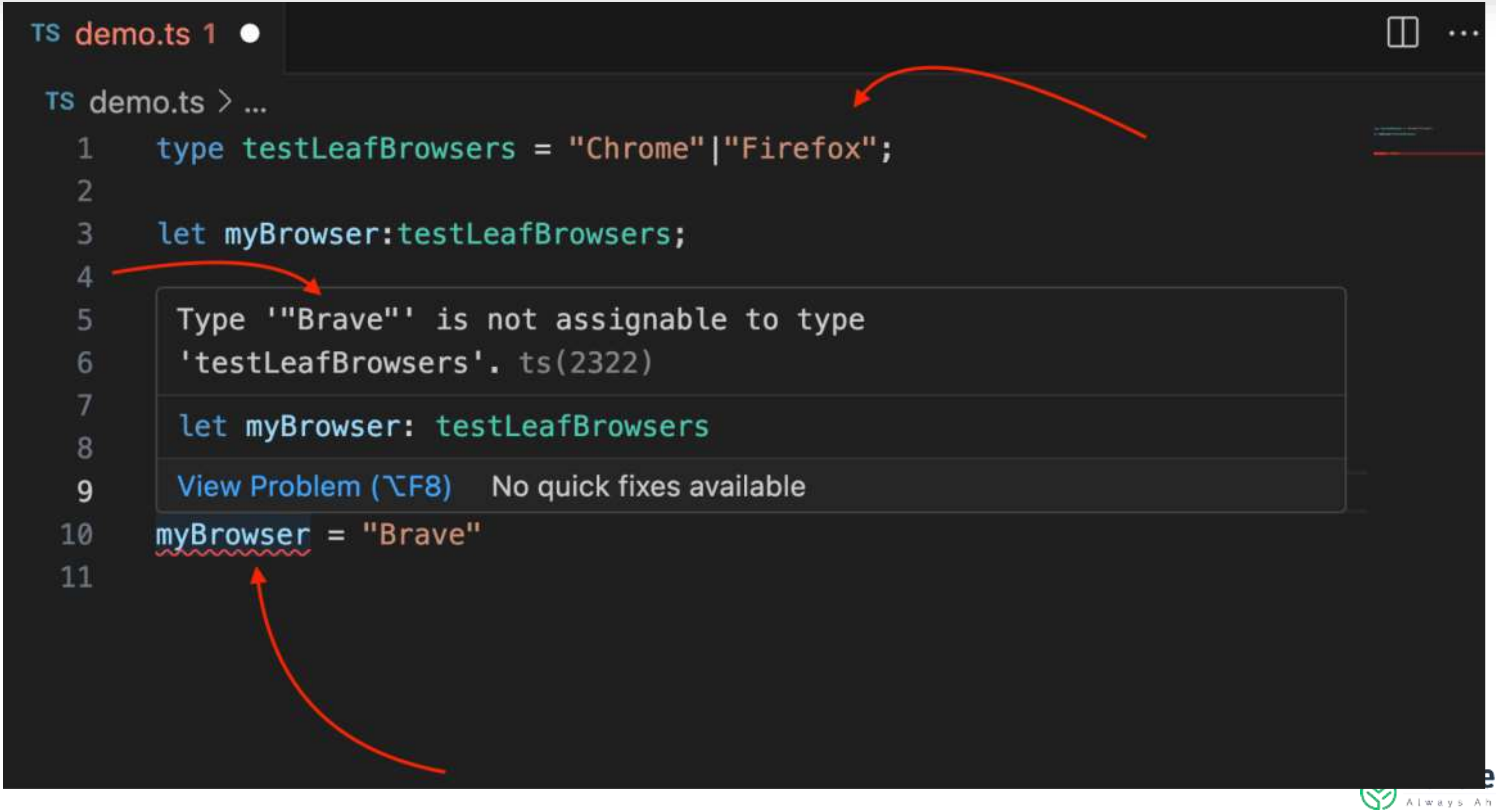
```
let myname = "prateek";
```

Explicit Types: We have to exactly to know and tell what kind of type the value is



```
let myname:string
myname = "prateek"
console.log(`my name is ${myname}`)
```

Custom Types in TS



```
type testLeafBrowsers = "Chrome"|"Firefox";

let myBrowser:testLeafBrowsers;







myBrowser = "Brave"

```

Type '"Brave"' is not assignable to type 'testLeafBrowsers'. ts(2322)

```
let myBrowser: testLeafBrowsers
```