FRONTEND FRAMEWORK

ANGULAR

INTRO TO FRAMEWORK AND ANGULAR

A software framework is a platform that provides generic functionalities that developers can customize according to the requirements of their app.

A software framework provides:

- A reusable environment
- A broad generic structure for your apps

FRAMEWORK VS LIBRARY

Framework	Library
Application code calls for specific functionality.	Library provides a set of helper functions, objects, or modules.
Framework calls your code.	Your code calls the library.
When a framework calls your code, the control is inverted.	When you call a method in a library, you are in control.

ANGULAR

Angular is a framework for building front-end applications.

It's an open-source, TypeScript-based framework.

It was developed in 2009 by Misko Hevery.

It is currently maintained by Google.

Angular apps are generally SPAs and can run on desktop as well as mobile devices.

MODULE 1

- Introduction to SPA
- What do we need Angular for?
- Setting up the Development Environment.
- Angular Application Architecture
- Angular CLI
- Project Structure

INTRODUCTION TO SPA

- A Single Page Application is a web application or website that interacts with the user by dynamically rewriting the current page, rather than loading entire new pages from the server.
- This approach provides a more fluid and fast user experience, similar to a desktop application.
- SPAs are built using modern JavaScript frameworks like Angular, React, and Vue.js.
- Examples:
  - Gmail
  - Google Maps
  - Facebook
  - Twitter
  - Instagram

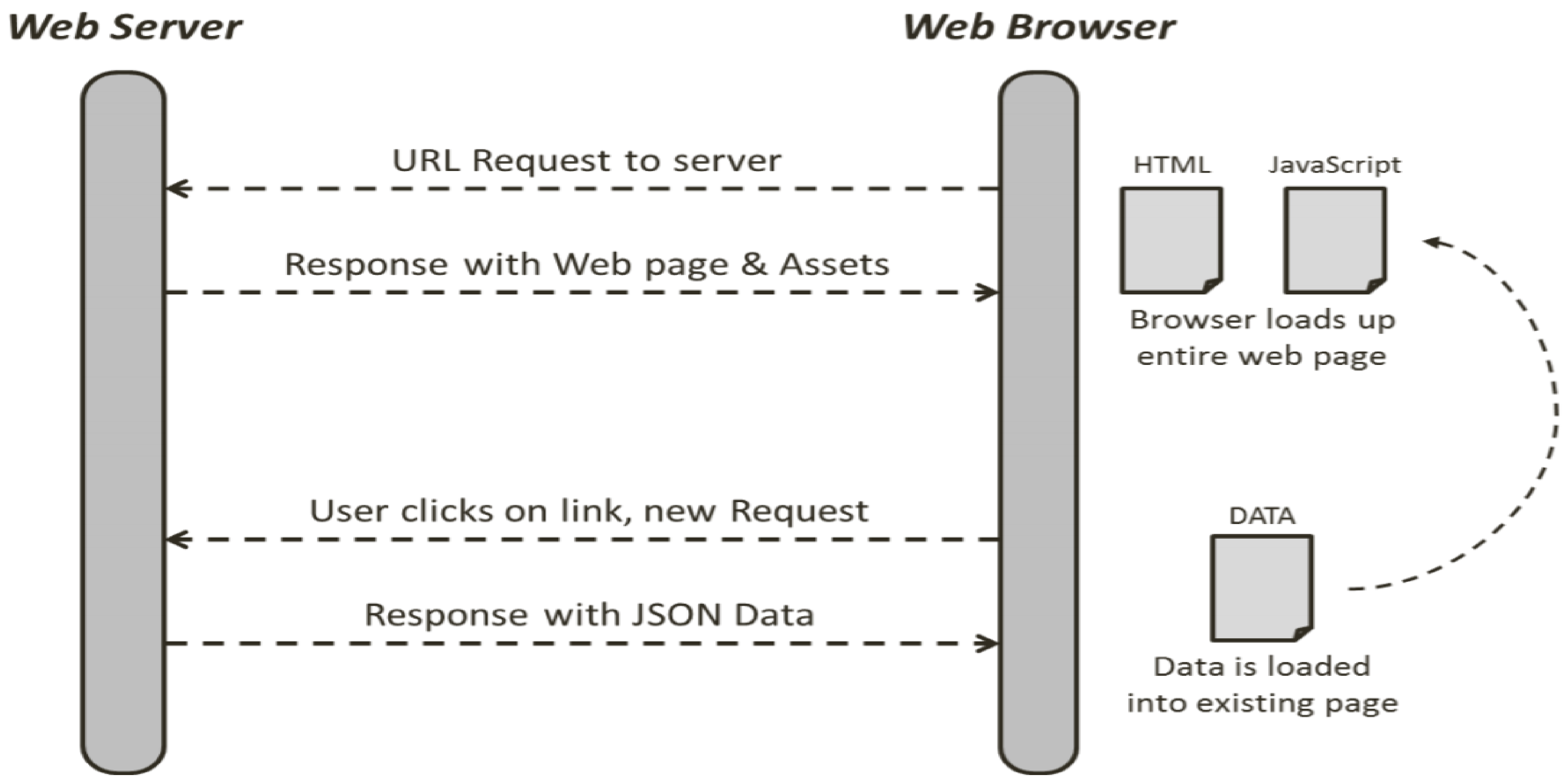
WHY ANGULAR?

- Angular is a popular front-end framework that helps developers build **Single Page Applications (SPAs)** and complex web applications.
- It provides a comprehensive set of tools, features, and architecture that make the development of dynamic, interactive, and scalable applications easier and more efficient.
- Features:
  - Building Single Page Applications (SPAs)
  - Component-Based Architecture
  - Two-Way Data Binding
  - Modular Development
  - Dependency Injection
  - Routing and Navigation
  - Reactive Forms and Template-Driven Forms
  - TypeScript Support

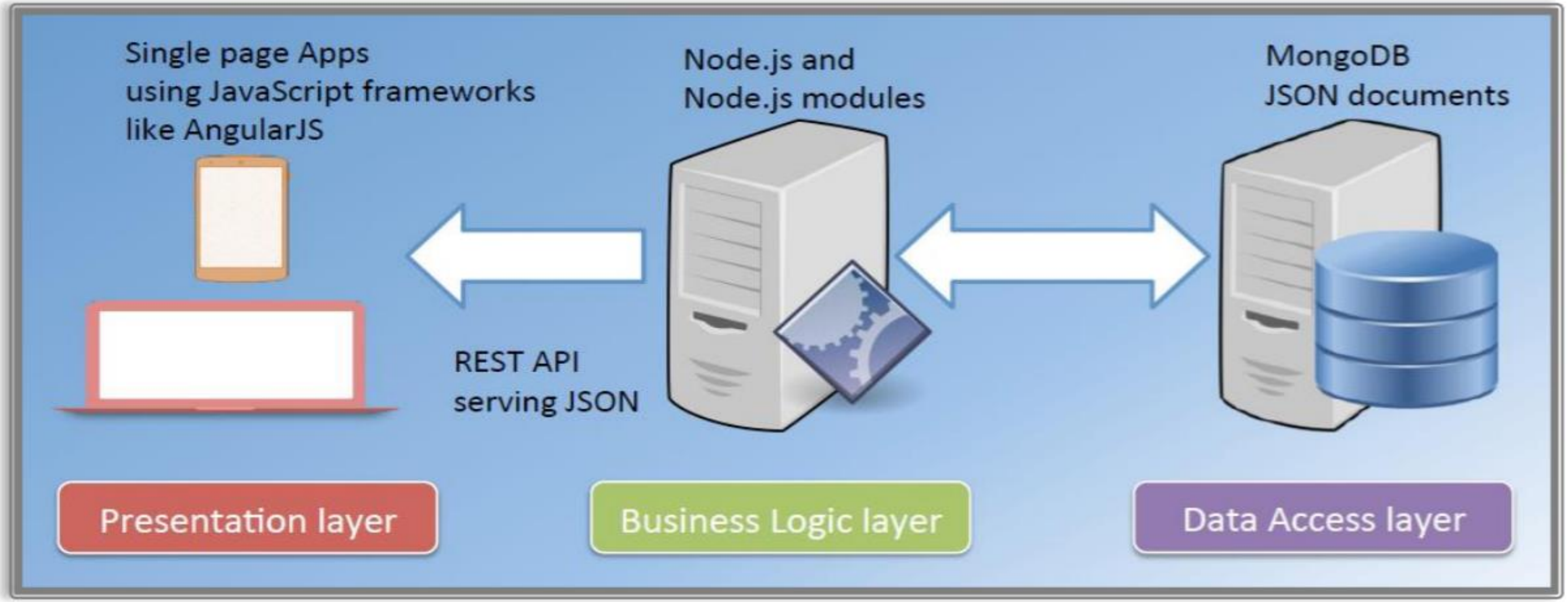
SET UP AND ENVIRONMENT

- Install Node.js: check installed node JS version by using below command:
  - node -v
  - npm -v
- Install Angular CLI:
  - npm install -g @angular/cli@15
- Check Version:
  - Ng version
- Create a New Angular 15 Project
  - ng new my-angular-app
- Navigate to the project folder
  - cd my-angular-app
- Run the project:
  - ng serve
- Once compiled, open a browser and go to:
  - http://localhost:4200

HOW APP WORKS?


Web Server
Web Browser
URL Request to server
Response with Web page & Assets
HTML
JavaScript
Browser loads up entire web page
User clicks on link, new Request
Response with JSON Data
DATA
Data is loaded into existing page

ROLE OF ANGULAR



PROJECT STRUCTURE

MODULE 2

- What is TypeScript
- Basic Types in TypeScript
- Variable Declarations using Let and Const
- Spread and Destructure.
- Classes
- Interfaces
- Arrow Functions
- Modules
- fetch API
- async / await

INTRODUCTION TO TYPESCRIPT

- TypeScript is the superset of JavaScript.
- Any valid JavaScript code is also a valid TypeScript code.
- TypeScript offers more features as compared to vanilla JavaScript.
- TypeScript does not run in the browser but is compiled to JavaScript by CLI.
- TypeScript is the main language for using Angular.
- Most examples and documentations of Angular use TypeScript.

FEATURES

- More predictable
- Reduced compile-time errors
- IDE supported
- Possesses object-oriented features
- Provides Classes, Interfaces, Modules, Properties, Generics, and Import and Export functionalities
- Easy to debug

INSTALLATION & TRANSPILING

- Run the command below to install TypeScript:
  - npm install –g typescript
- Run the command below to transpile TypeScript to JavaScript:
  - tsc some-program.ts
  - It will convert .ts file to .js which we can run using node command
  - Node some-program.js

INTRODUCTION TO DATATYPE

- TypeScript supports the following data types:
    - Boolean
    - Number
    - String
    - Array
    - Any
    - Tuple
    - Enum
    - Void

LET VS CONST

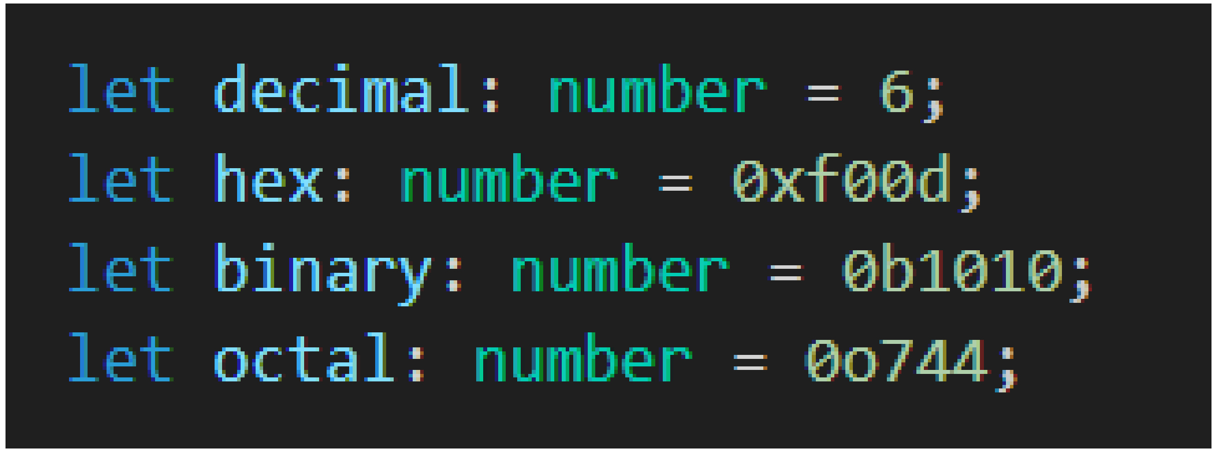
- There are two keywords used to declare variables in TypeScript: *let* and *const*
- *let* declares a block-scoped local variable
- *const* defines block-scoped constants whose values cannot change through re-assignment
- Variables declared using *const* can't be redeclared

```
const name = 'Harry';
let age = 25;
const dateOfBirth = '10/10/2017';
```

VARIABLE DECLARATION

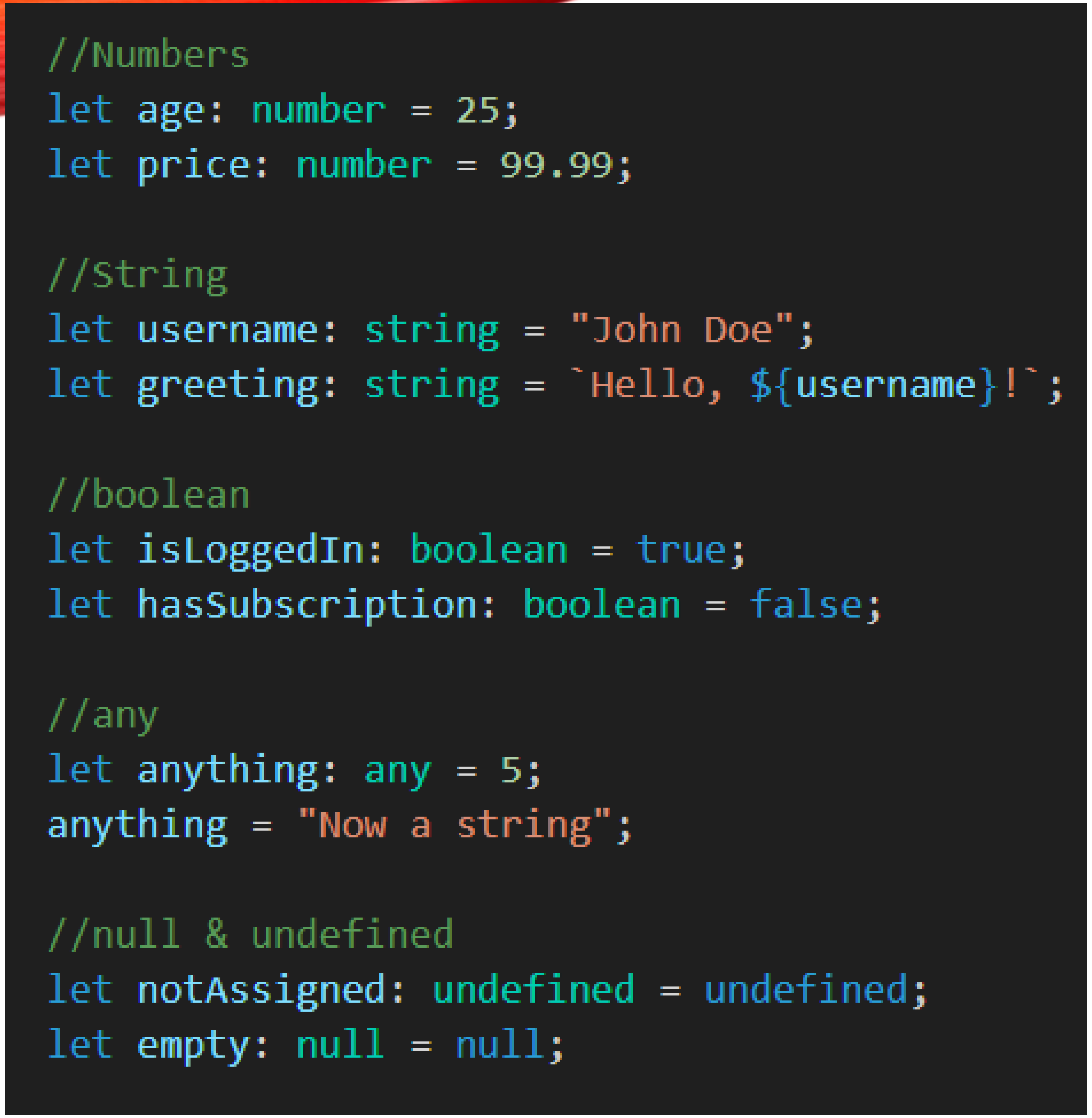
let <variable-name>: <data-type> = <value>;

const <variable-name>: <data-type> = <value>;

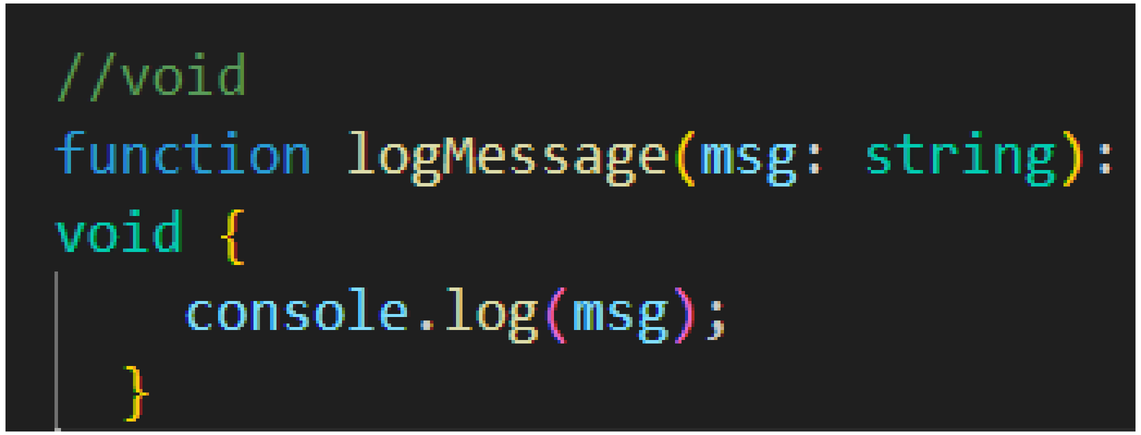


```
let decimal: number = 6;
let hex: number = 0xf00d;
let binary: number = 0b1010;
let octal: number = 0o744;
```

SOME EXAMPLES



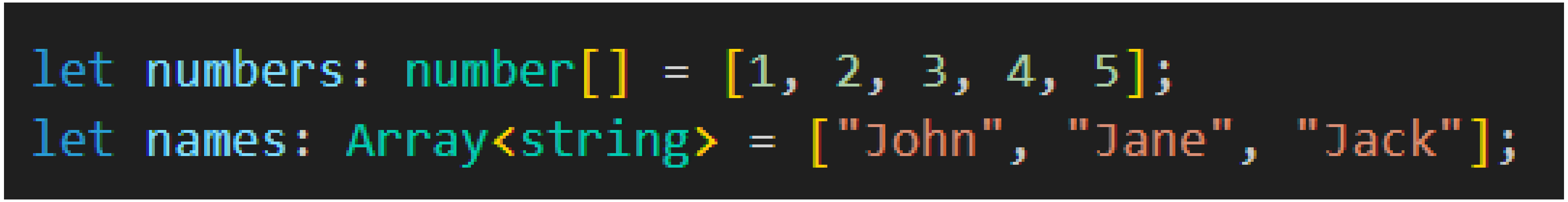
```
//Numbers
let age: number = 25;
let price: number = 99.99;

//String
let username: string = "John Doe";
let greeting: string = `Hello, ${username}!`;

//boolean
let isLoggedIn: boolean = true;
let hasSubscription: boolean = false;

//any
let anything: any = 5;
anything = "Now a string";

//null & undefined
let notAssigned: undefined = undefined;
let empty: null = null;
```



```
//void
function logMessage(msg: string):
void {
    console.log(msg);
  }
```

ARRAY

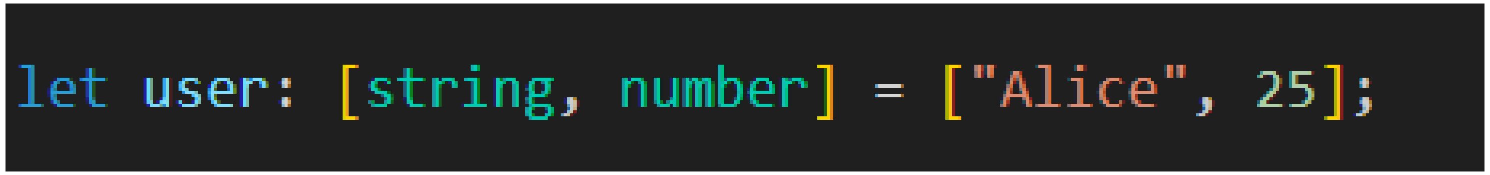
- TypeScript arrays can hold a collection of values of a specific type.
- You can declare arrays in two ways:
  - using square brackets []
  - using Array<type>



```
let numbers: number[] = [1, 2, 3, 4, 5];
let names: Array<string> = ["John", "Jane", "Jack"];
```

TUPLES

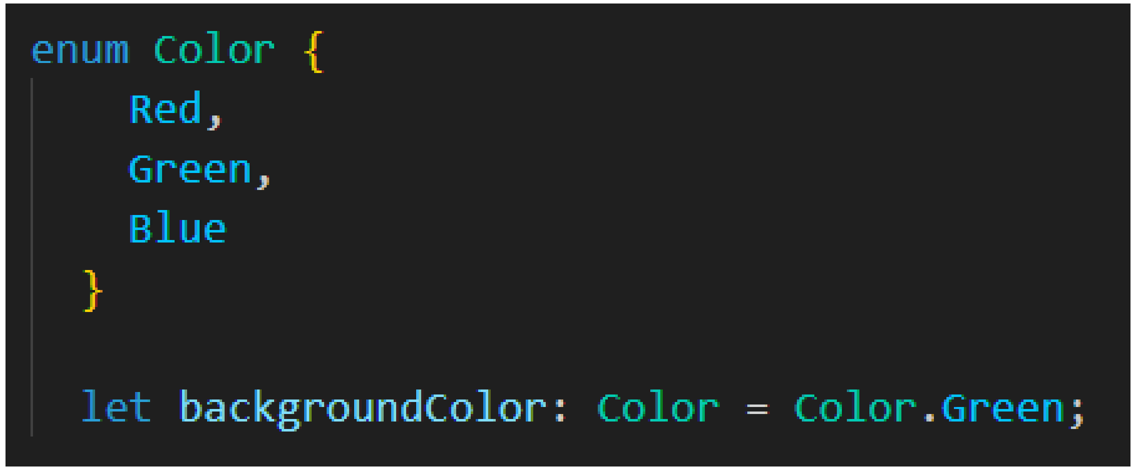
- A tuple is an array with fixed sizes and types for each index.
- It allows different types of values in each position.



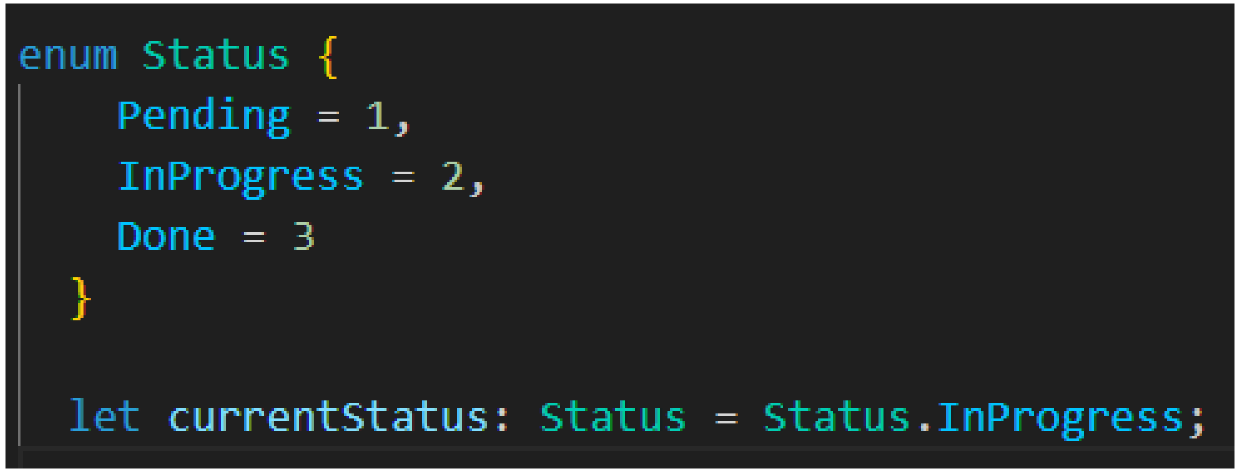
```
let user: [string, number] = ["Alice", 25];
```

ENUM

- enum is a special data type that defines a set of named constants.
- It allows you to create a collection of related values.



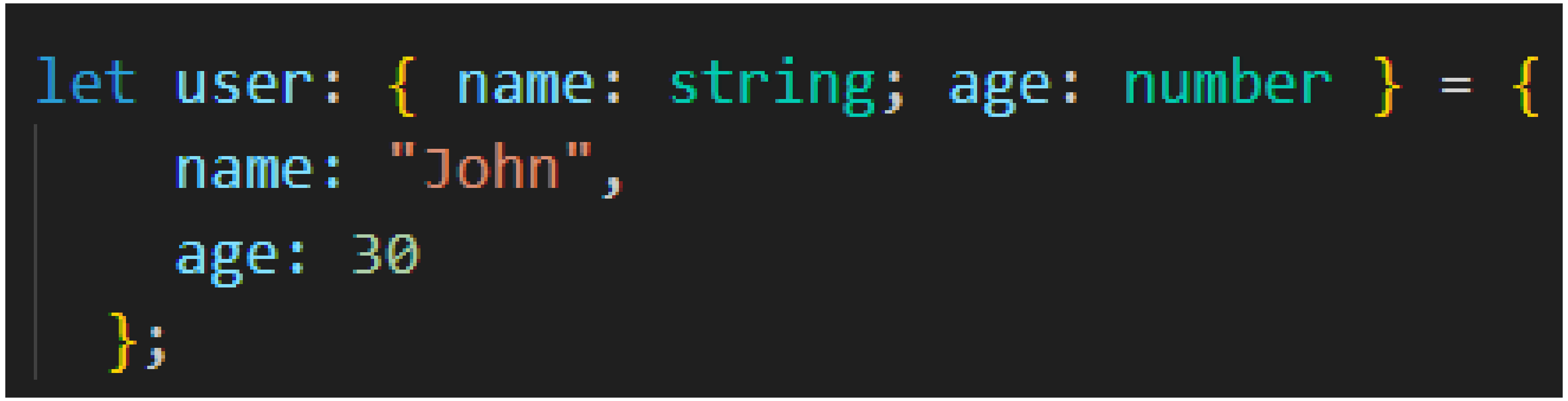
```
enum Color {
    Red,
    Green,
    Blue
  }

  let backgroundColor: Color = Color.Green;
```



```
enum Status {
    Pending = 1,
    InProgress = 2,
    Done = 3
  }

  let currentStatus: Status = Status.InProgress;
```

ONJECTS

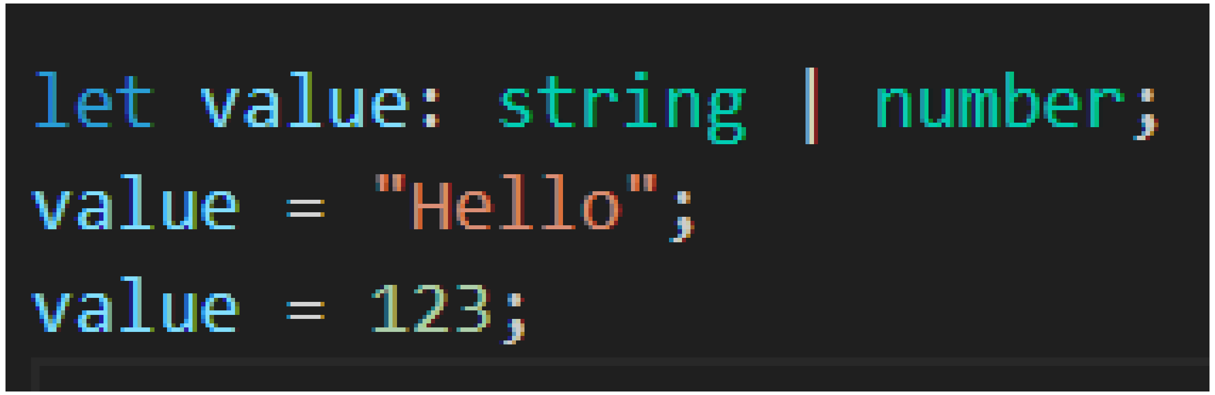
- Objects in TypeScript can be defined with a specific structure (type).
- Each property can have a type assigned to it.



```
let user: { name: string; age: number } = {
    name: "John",
    age: 30
  };
```

UNION TYPES

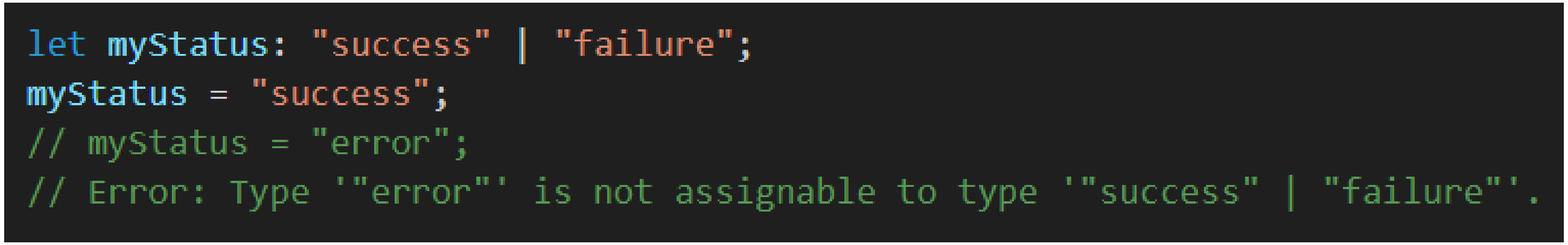
- A union type allows a variable to have multiple types.
- It's defined using the pipe (|) operator.



```
let value: string | number;
value = "Hello";
value = 123;
```

LITERAL TYPES

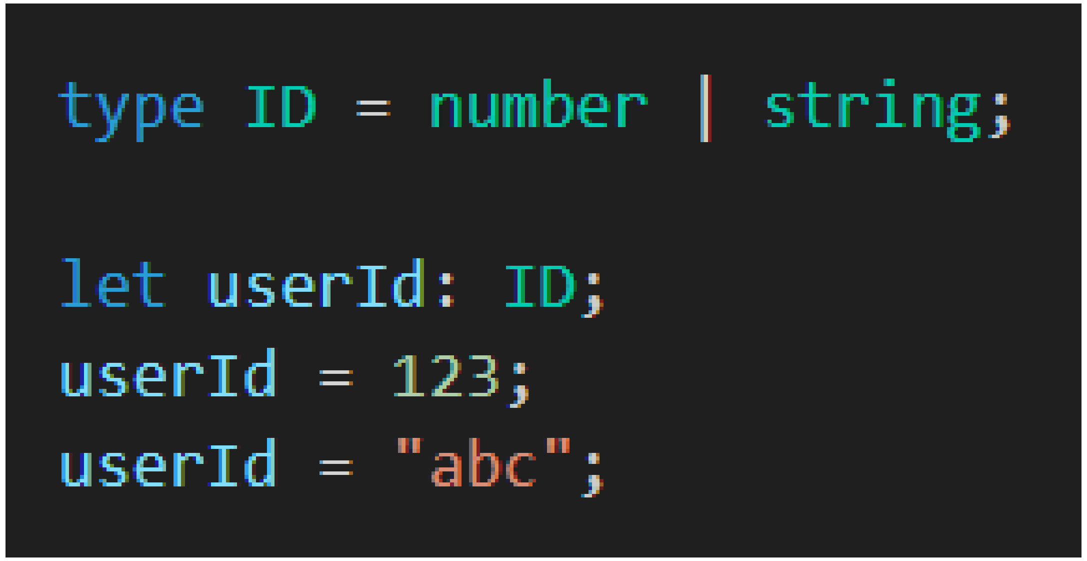
- Literal types allow you to specify an exact value a variable can take.



```
let myStatus: "success" | "failure";
myStatus = "success";
// myStatus = "error";
// Error: Type '"error"' is not assignable to type '"success" | "failure"'.
```

TYPE ALIASES

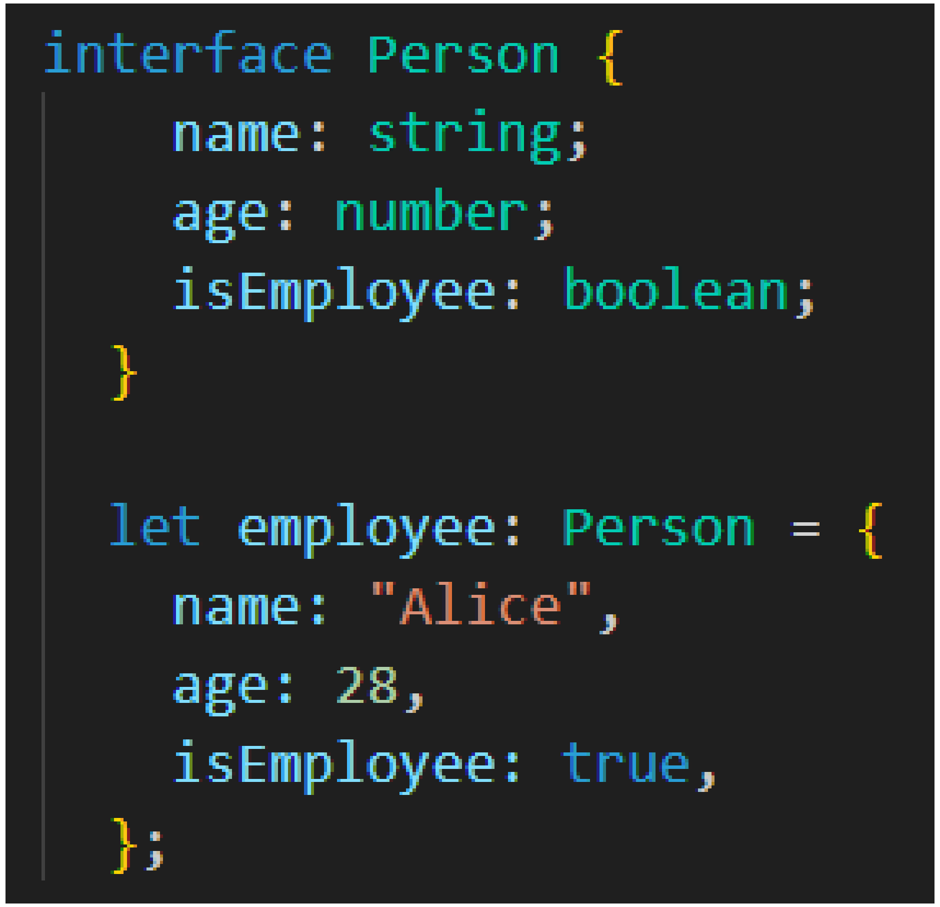
- You can create custom type aliases to make code more readable.



```
type ID = number | string;

let userId: ID;
userId = 123;
userId = "abc";
```

INTERFACES

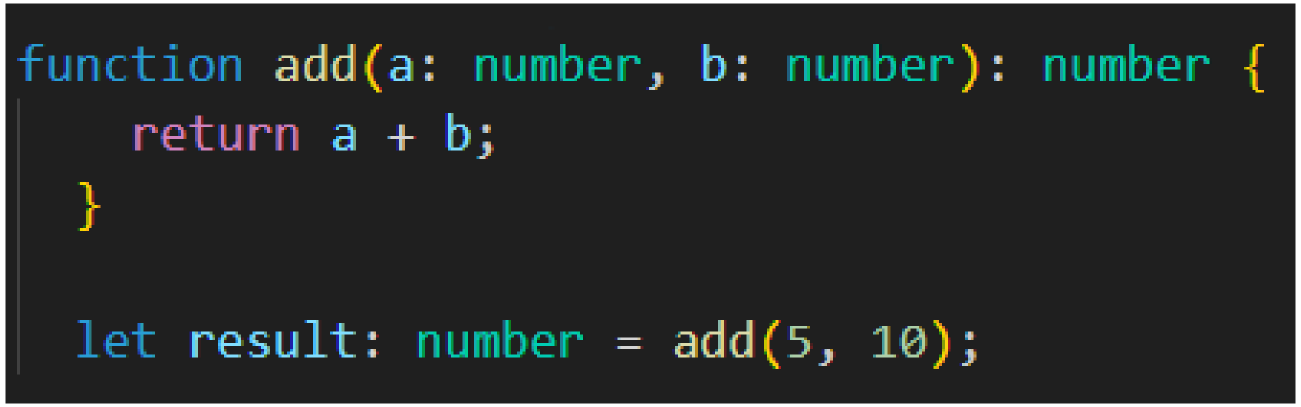
- Interfaces define the structure of an object.
- Unlike type, they are mainly used to define object types.



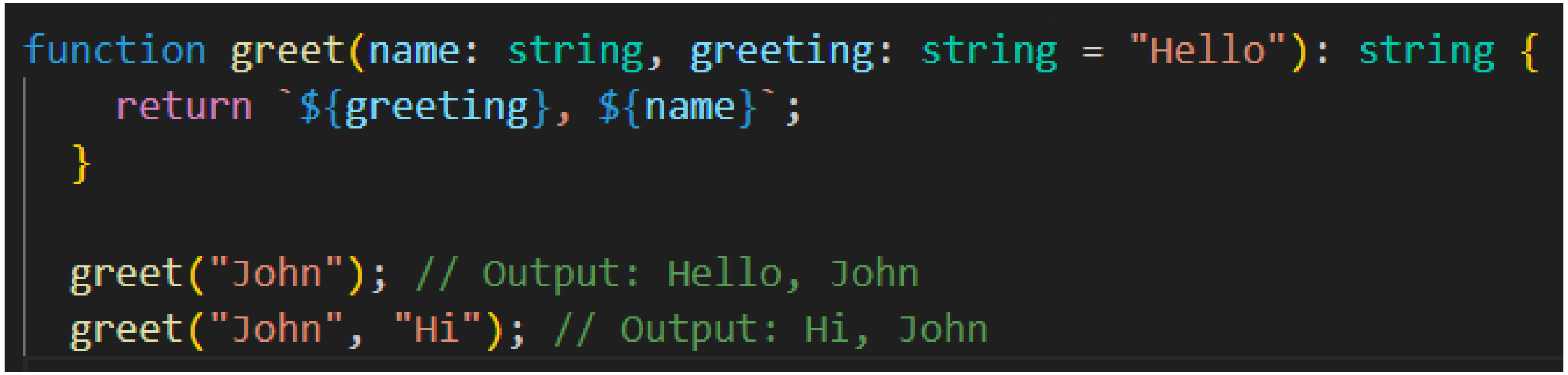
```
interface Person {
    name: string;
    age: number;
    isEmployee: boolean;
  }

  let employee: Person = {
    name: "Alice",
    age: 28,
    isEmployee: true,
  };
```

FUNCTIONS

- You can define function parameter and return types in TypeScript.

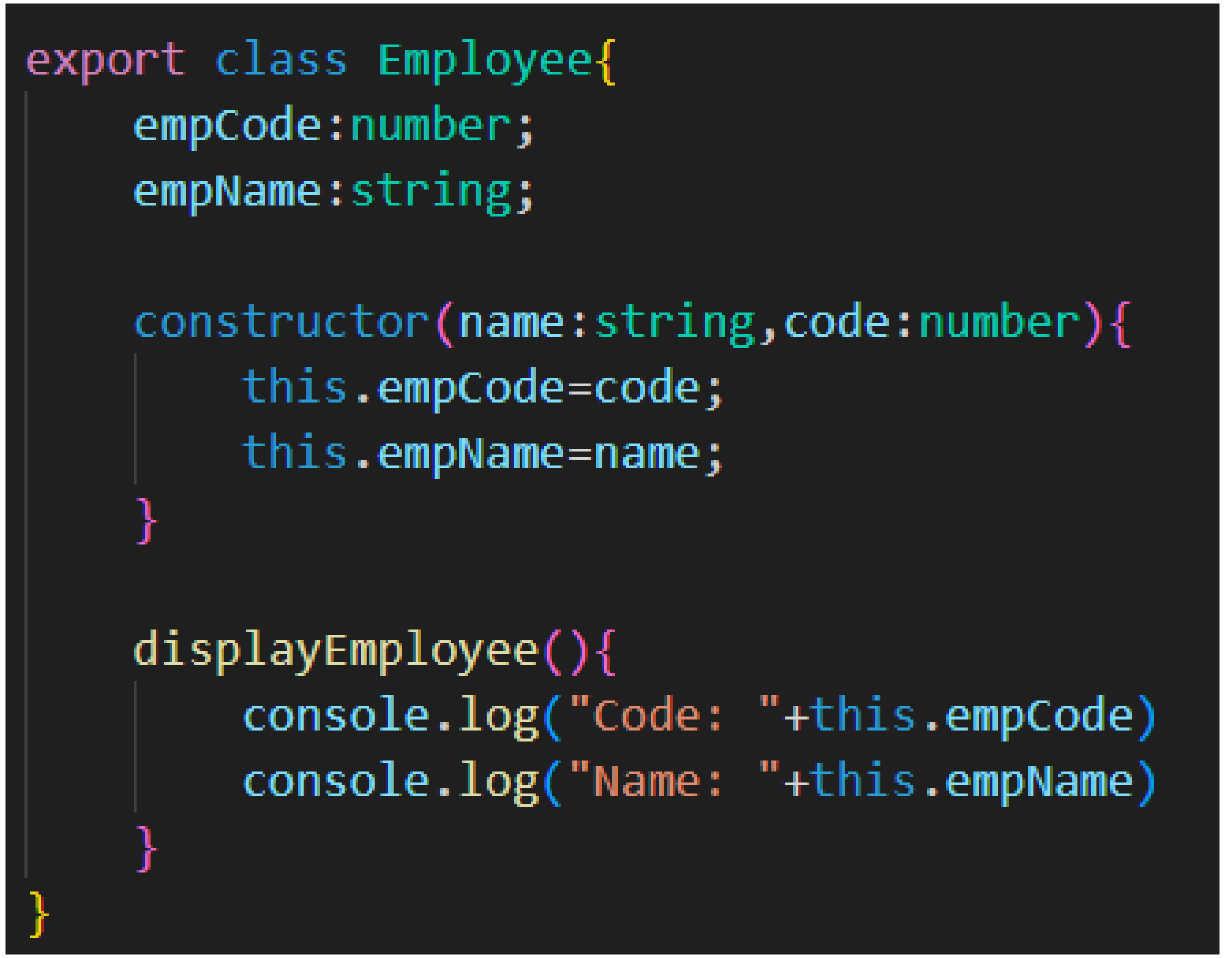


```
function add(a: number, b: number): number {
    return a + b;
  }

  let result: number = add(5, 10);
```



```
function greet(name: string, greeting: string = "Hello"): string {
    return `${greeting}, ${name}`;
  }

  greet("John"); // Output: Hello, John
  greet("John", "Hi"); // Output: Hi, John
```

CLASS



```
export class Employee{
    empCode:number;
    empName:string;

    constructor(name:string,code:number){
        this.empCode=code;
        this.empName=name;
    }

    displayEmployee(){
        console.log("Code: "+this.empCode)
        console.log("Name: "+this.empName)
    }
}
```

- It represent the structure or blue print of class which includes member variables, functions and constructor.
- From the given structure we can create as many objects as we want.

let emp1= new Employee("sonam",12);

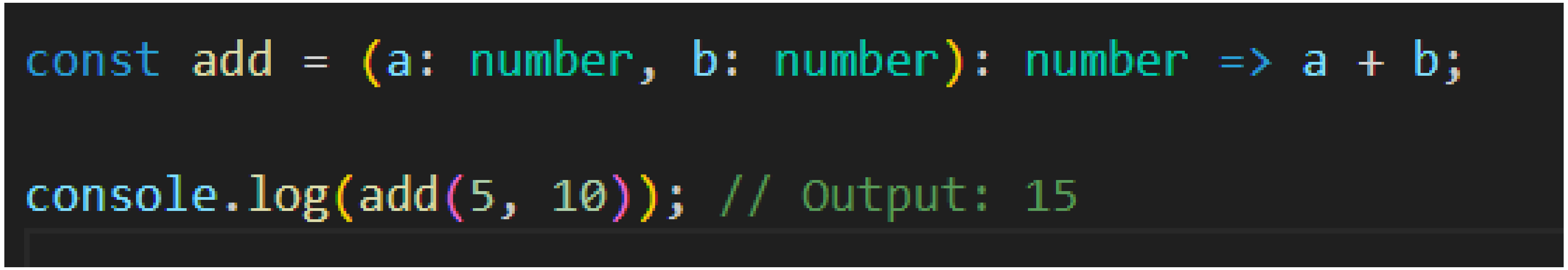
let emp2= new Employee("Akhshita",56);

emp1.displayEmployee();

emp2.displayEmployee();

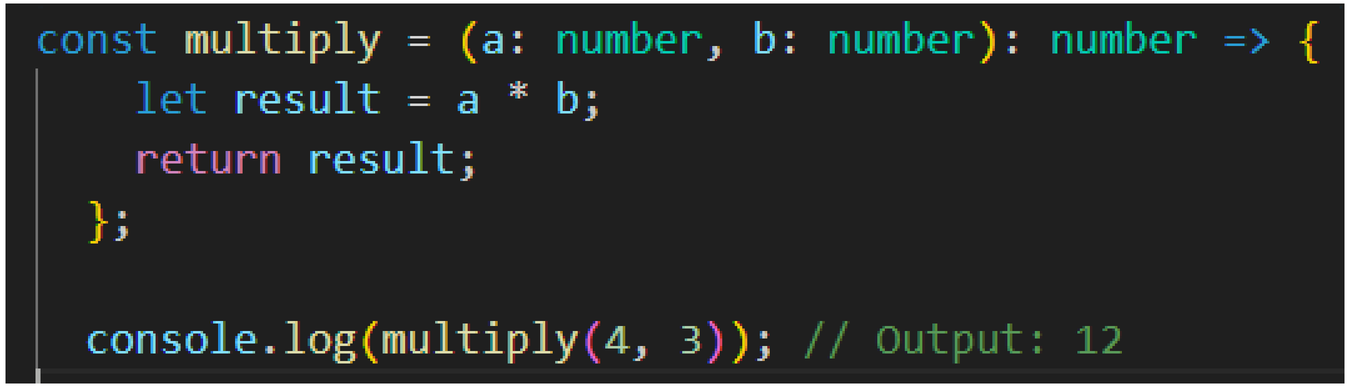
ARROW FUNCTIONS

- In TypeScript, **arrow functions** are a more concise syntax for writing functions, introduced in ES6.
- Arrow functions are often used because they simplify function expressions and have a different behavior from regular functions when it comes to this binding, making them useful in many contexts.



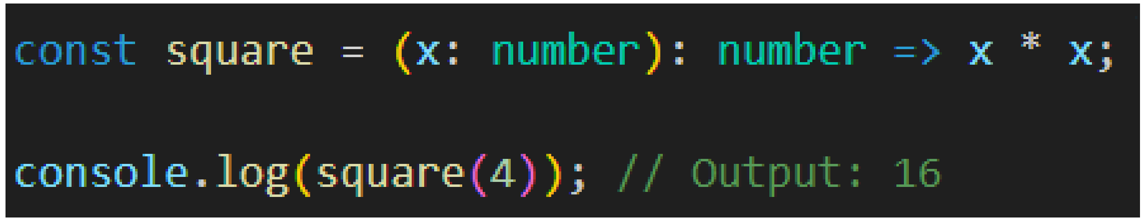
```
const add = (a: number, b: number): number => a + b;

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

Multiple Parameters



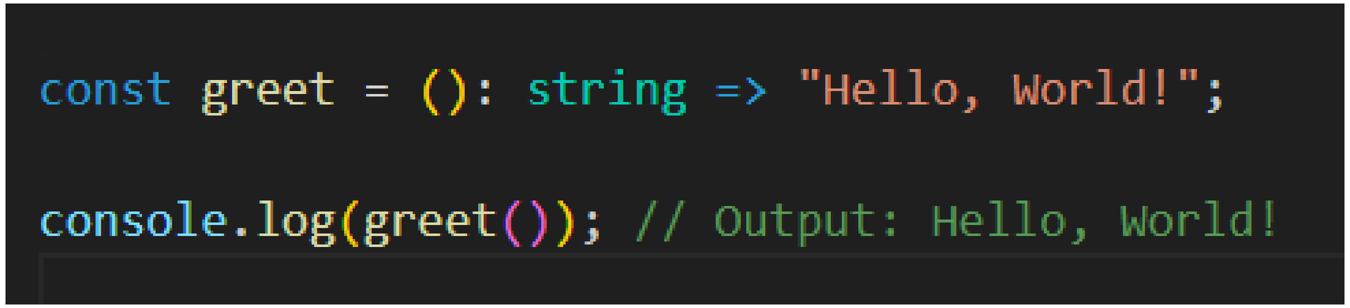
```
const multiply = (a: number, b: number): number => {
    let result = a * b;
    return result;
 };

 console.log(multiply(4, 3)); // Output: 12
```

One Parameters



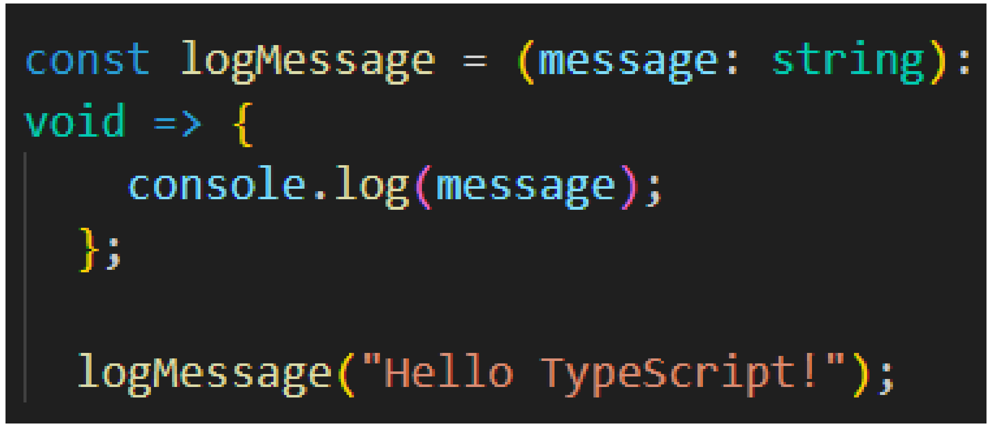
```
const square = (x: number): number => x * x;

console.log(square(4)); // Output: 16
```

No Parameters



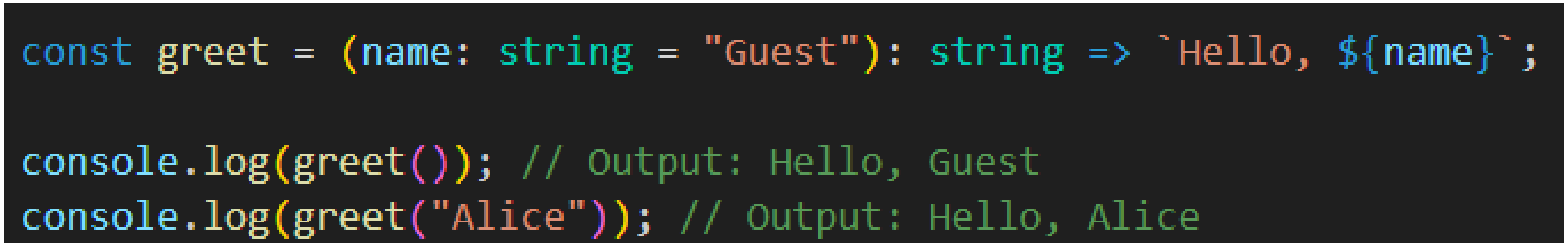
```
const greet = (): string => "Hello, World!";

console.log(greet()); // Output: Hello, World!
```

Void Function



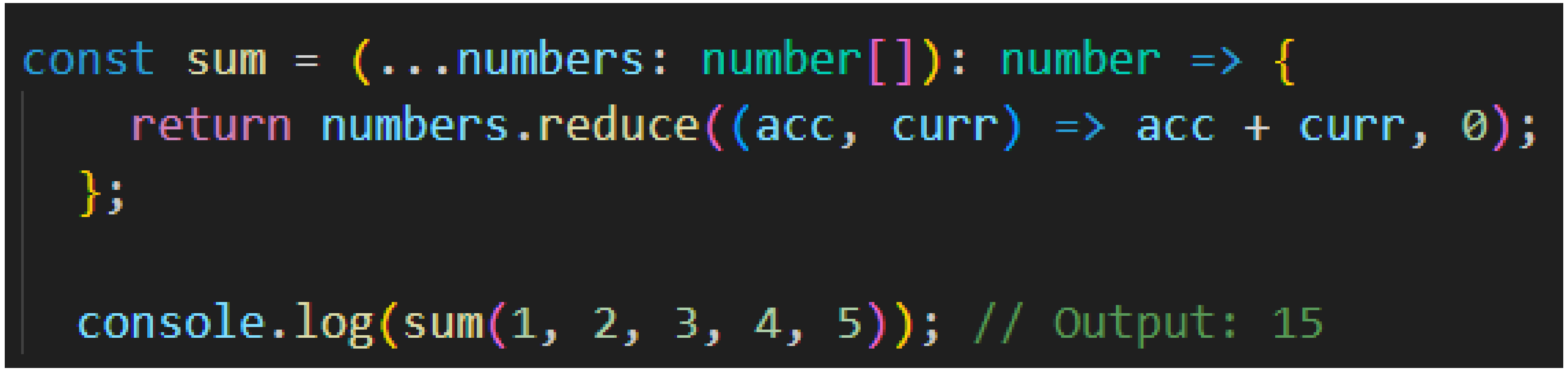
```
const logMessage = (message: string):
void => {
    console.log(message);
  };

  logMessage("Hello TypeScript!");
```

Default Parameters



```
const greet = (name: string = "Guest"): string => `Hello, ${name}`;

console.log(greet()); // Output: Hello, Guest
console.log(greet("Alice")); // Output: Hello, Alice
```

REST Parameters



```
const sum = (...numbers: number[]): number => {
    return numbers.reduce((acc, curr) => acc + curr, 0);
  };

  console.log(sum(1, 2, 3, 4, 5)); // Output: 15
```

ADVANTAGES OF ARROW FUNCTIONS

Conciseness:

- Arrow functions provide a shorter syntax, which makes code more readable.

Lexical this Binding:

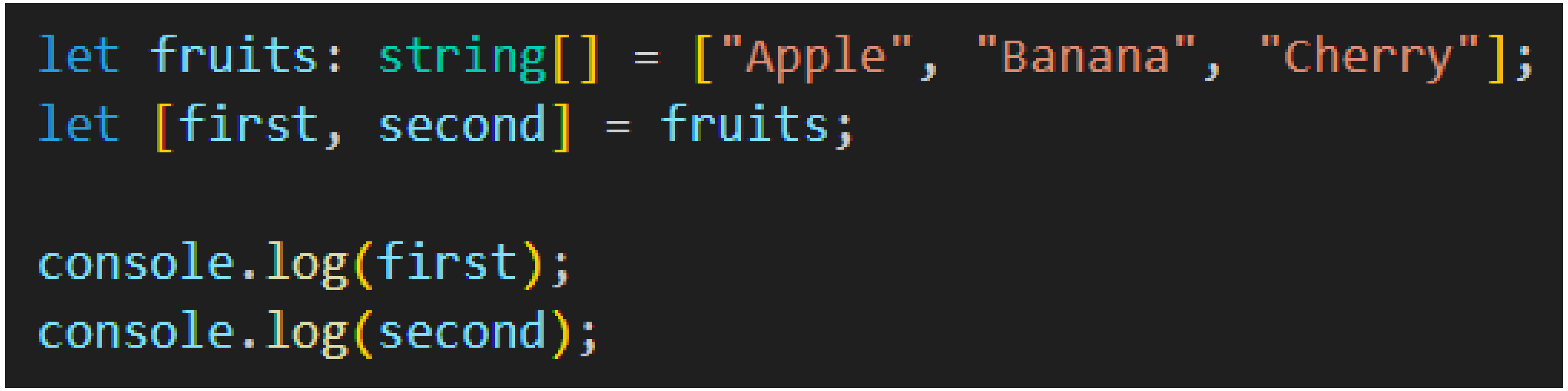
- They automatically bind this to the surrounding context, preventing common errors in callbacks or event handlers.

Implicit Returns:

- If the arrow function has only one expression, you can omit the return statement, making it cleaner.

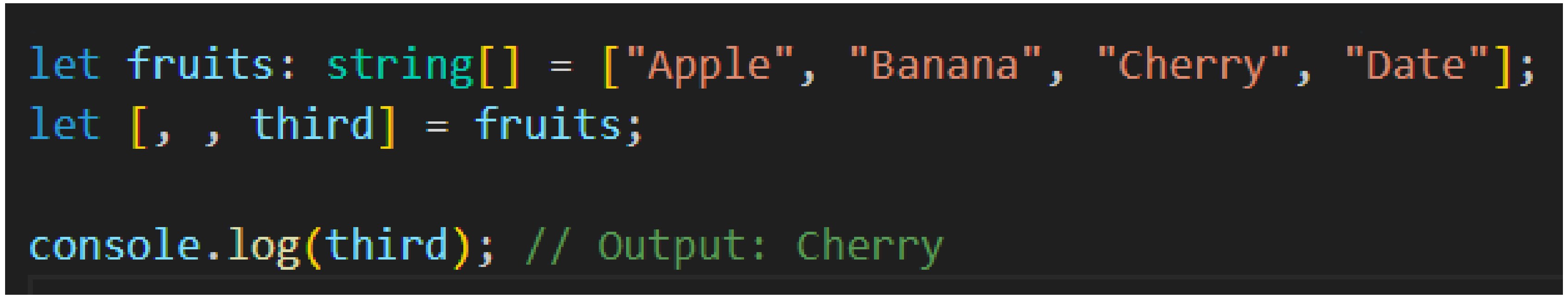
DESTRUCTURING

- **Destructuring** in TypeScript is a convenient way to extract values from arrays and objects and assign them to variables.
- It is a syntax feature from ES6, fully supported in TypeScript, that allows unpacking data into distinct variables for easier manipulation.



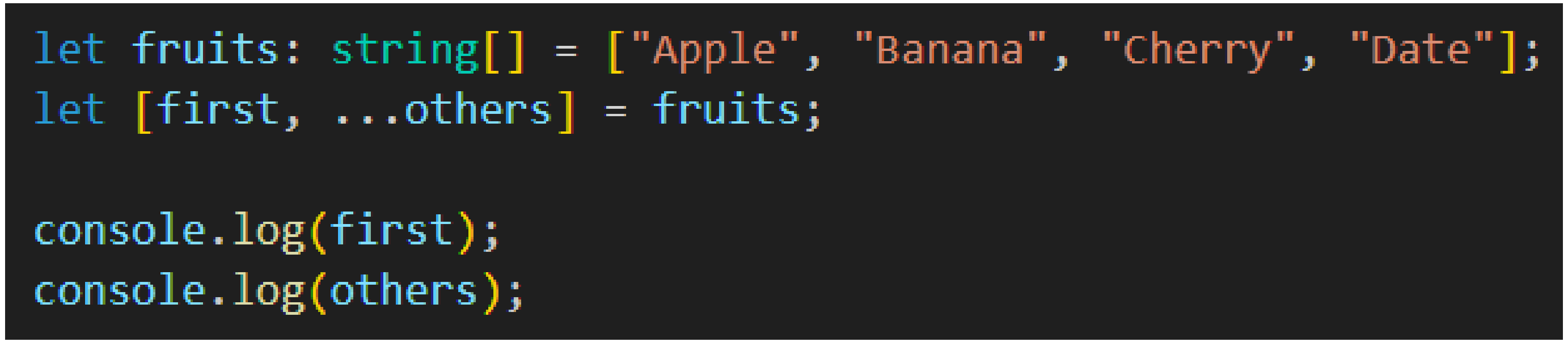
```
let fruits: string[] = ["Apple", "Banana", "Cherry"];
let [first, second] = fruits;

console.log(first);
console.log(second);
```

Element Skipping



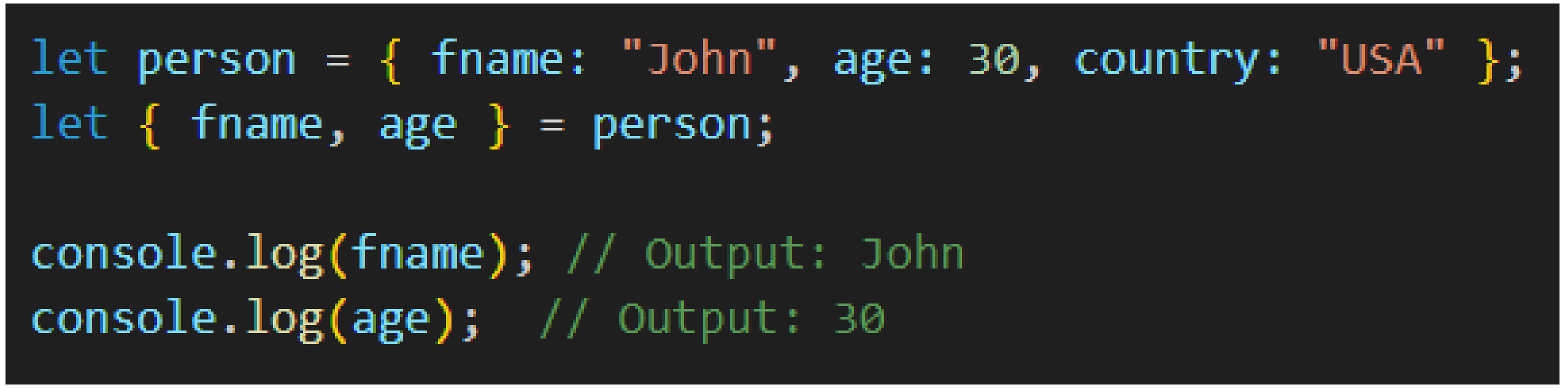
```
let fruits: string[] = ["Apple", "Banana", "Cherry", "Date"];
let [, , third] = fruits;

console.log(third); // Output: Cherry
```

REST Operator



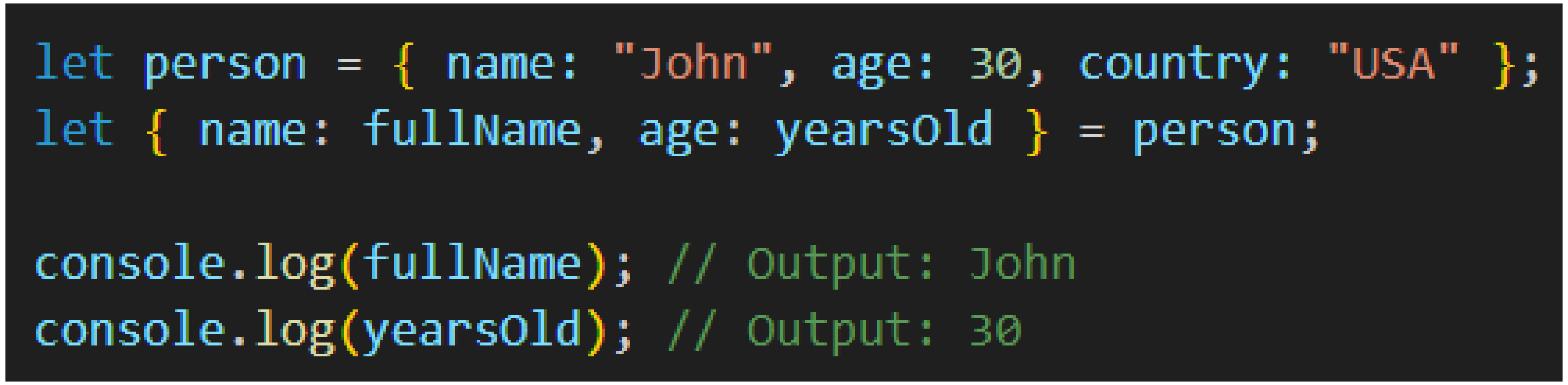
```
let fruits: string[] = ["Apple", "Banana", "Cherry", "Date"];
let [first, ...others] = fruits;

console.log(first);
console.log(others);
```

Object Destructuring



```
let person = { fname: "John", age: 30, country: "USA" };
let { fname, age } = person;

console.log(fname); // Output: John
console.log(age);  // Output: 30
```

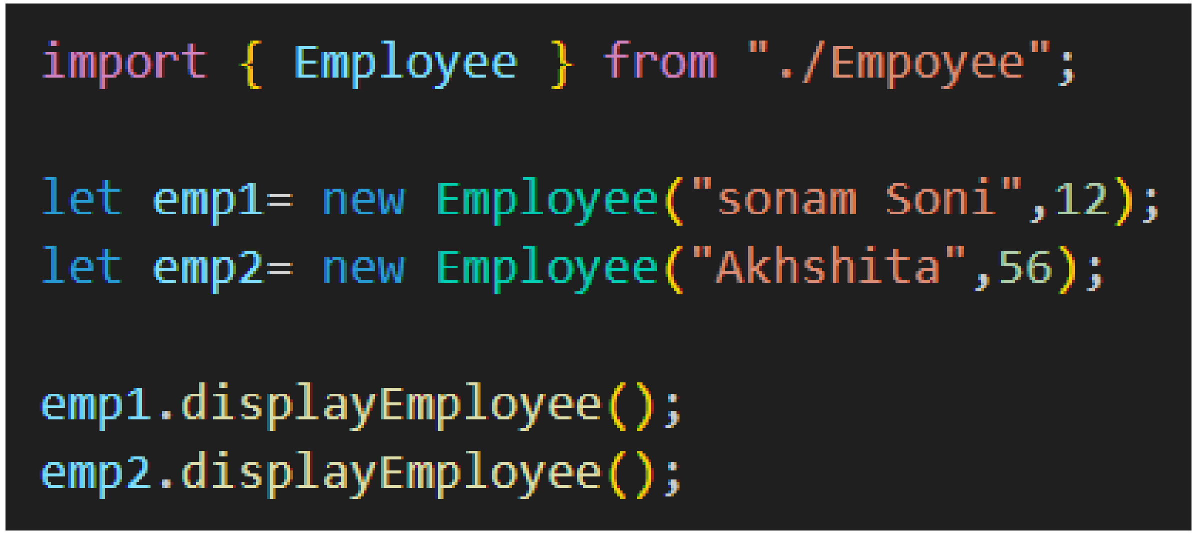
Variable rename



```
let person = { name: "John", age: 30, country: "USA" };
let { name: fullName, age: yearsOld } = person;

console.log(fullName); // Output: John
console.log(yearsOld); // Output: 30
```

MODULES

- We can create separate modules to use them into another file.
- To export use export keyword
- While using it to other file use import keyword.



```
import { Employee } from "./Empoyee";

let emp1= new Employee("sonam Soni",12);
let emp2= new Employee("Akhshita",56);

emp1.displayEmployee();
emp2.displayEmployee();
```

FETCH API & ASYNC AWAIT

TASK: CREATE A LIBRARY MANAGEMENT SYSTEM IN TYPESCRIPT

- **Book Class**
- title: string
- author: string
- isbn: string
- availableCopies: number

- **Library Class**:
- A property books (an array of Book objects).
- A method addBook(book: Book) to add a new book to the library.
- A method removeBook(isbn: string) to remove a book by ISBN.
- A method findBookByTitle(title: string) to find a book by title.
- A method borrowBook(isbn: string) that checks if a book is available and decreases availableCopies by 1.
- A method returnBook(isbn: string) that increases availableCopies by 1.

- **User Class**:
- name: string
- email: string
- borrowedBooks: an array of Book objects (the books borrowed by the user).
- Methods to borrow and return books from the library.

- **Instruction**:
- **Step 1**: Define the Book class with properties and constructor to initialize the book's details.
- **Step 2**: Create the Library class with methods to add, remove, borrow, and return books.
- **Step 3**: Define the User class with properties and methods to manage borrowed books.
- **Step 4**: Instantiate a library and add some books to it.
- **Step 5**: Instantiate a user and allow them to borrow and return books.

THE MAIN BUILDING BLOCKS

- Module
- Component
- Metadata
- Template
- Data Binding
- Service
- Directive
- Dependency Injection

BOOTSTRAPPING THE APP

- Import the bootstrap module
- Import your top-level component
- Import application dependencies
- Call bootstrap and pass in your top-level component as the first parameter and an array of dependencies as the second

MODULE

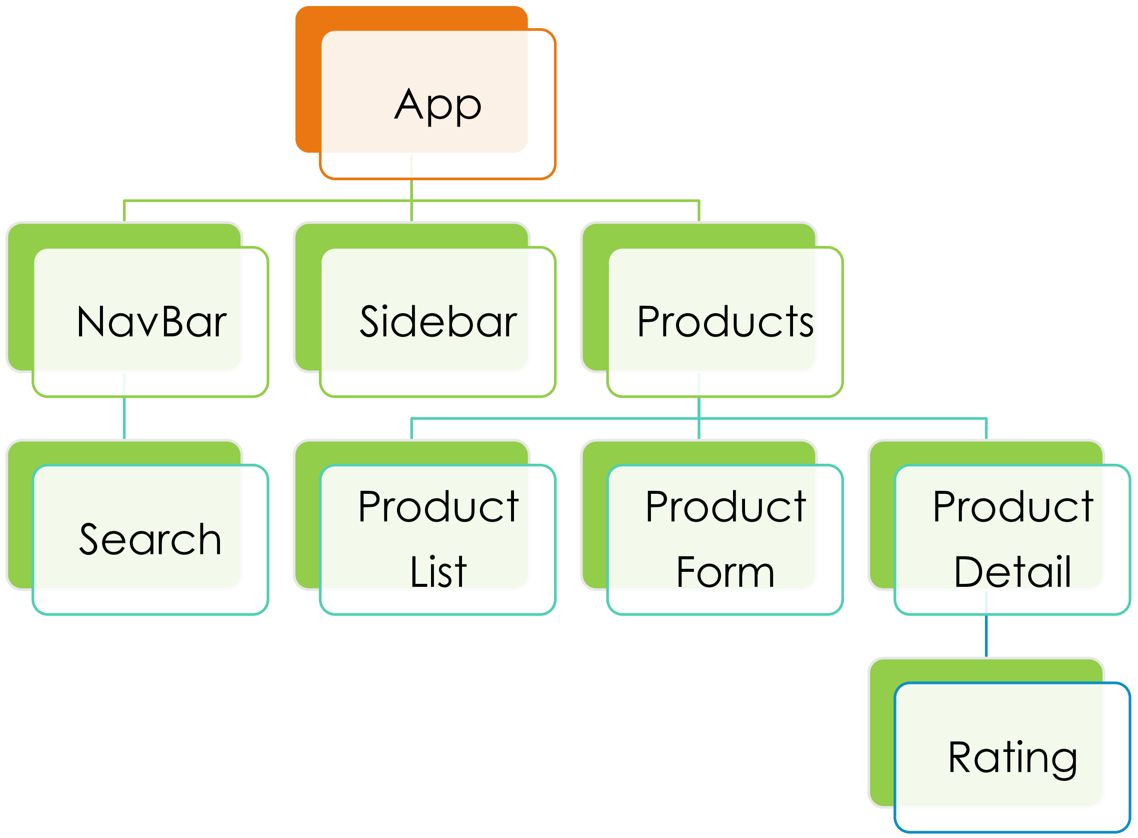
- **Modules** are containers for related components, directives, services, pipes, and other modules.
- Every Angular application has at least one module, called the **root module**, commonly named AppModule.
- **NgModule** is a decorator that defines metadata for the module (such as the components it uses, the services it provides, etc.).
- **Key Properties of NgModule:**
  - **declarations**: Specifies which components, directives, and pipes belong to this module.
  - **imports**: Allows you to bring in other modules (like FormsModule, HttpClientModule, etc.).
  - **providers**: Registers services that the module can use.
  - **bootstrap**: Defines the root component that Angular should bootstrap when starting the application.

COMPONENTS

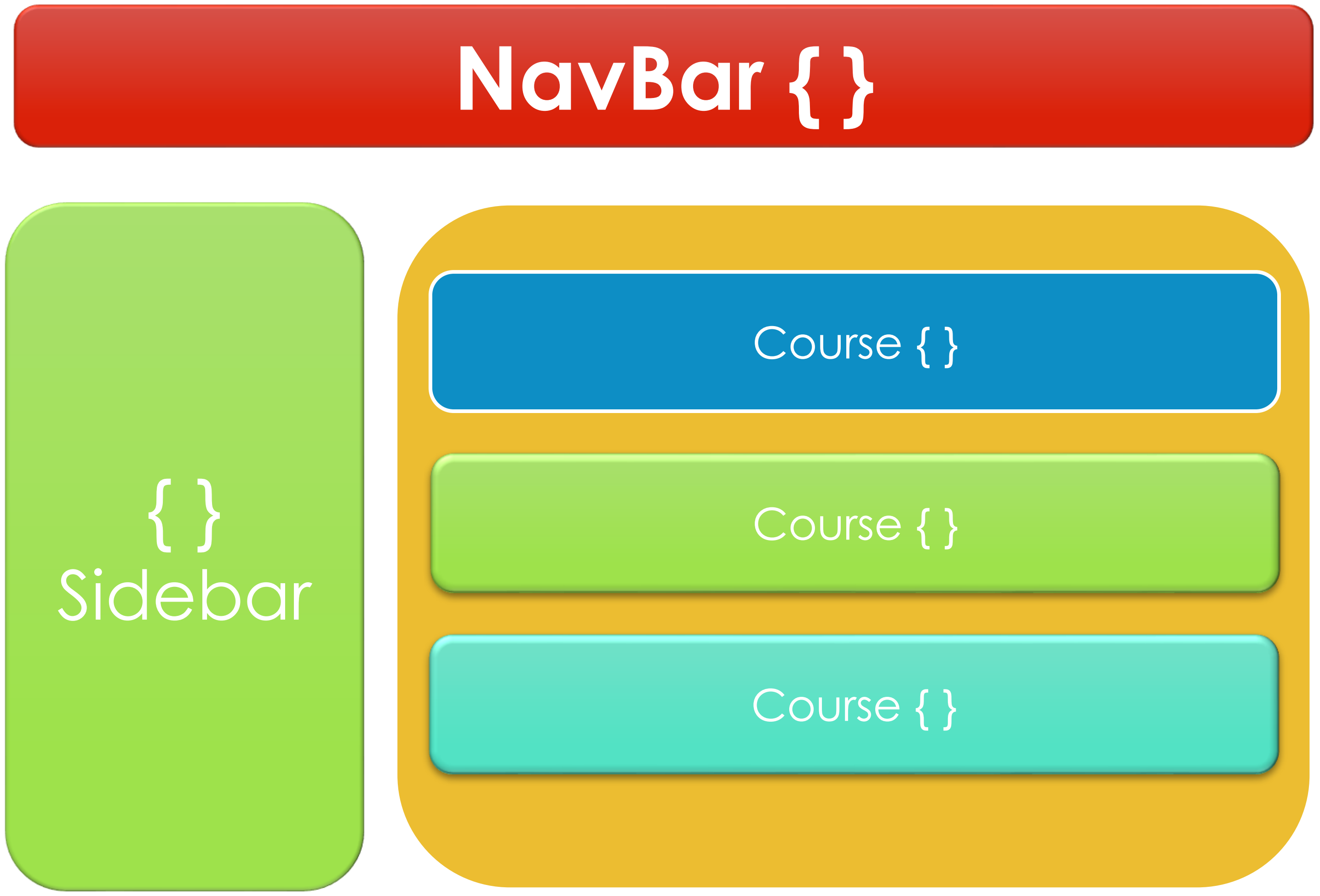
- A **Component** is the building block of an Angular application.
- It contains the **view** (HTML template), **logic** (TypeScript class), and **styles** (CSS) that define a part of the user interface.
- Components are defined using the @Component decorator, which specifies the template, styles, and the selector for the component.
- **Structure of a Component:**
  - **Selector**: A custom HTML tag that identifies the component.
  - **Template**: The HTML that defines the structure and presentation of the view.
  - **Class**: The logic behind the view, written in TypeScript. It defines properties, methods, and interactions.
  - **Styles**: Optional styles that apply to the component's template.

COMPONENTS

- Encapsulate the template, data, and the behavior of a view
- Allows you to break a complex web page into smaller, manageable, and reusable parts
- Plain Type Script class
- Promotes:
    - **Reusability**
    - **Maintainability**
    - **Testability**



COMPONENTS IN REAL APPLICATION



METADATA

- Metadata allows Angular to process a class
- We can attach metadata with TypeScript using decorators
- Decorators are just functions
- Most common is the @Component() decorator
- Takes a config option with the selector, template(Url), providers, directives, pipes and styles

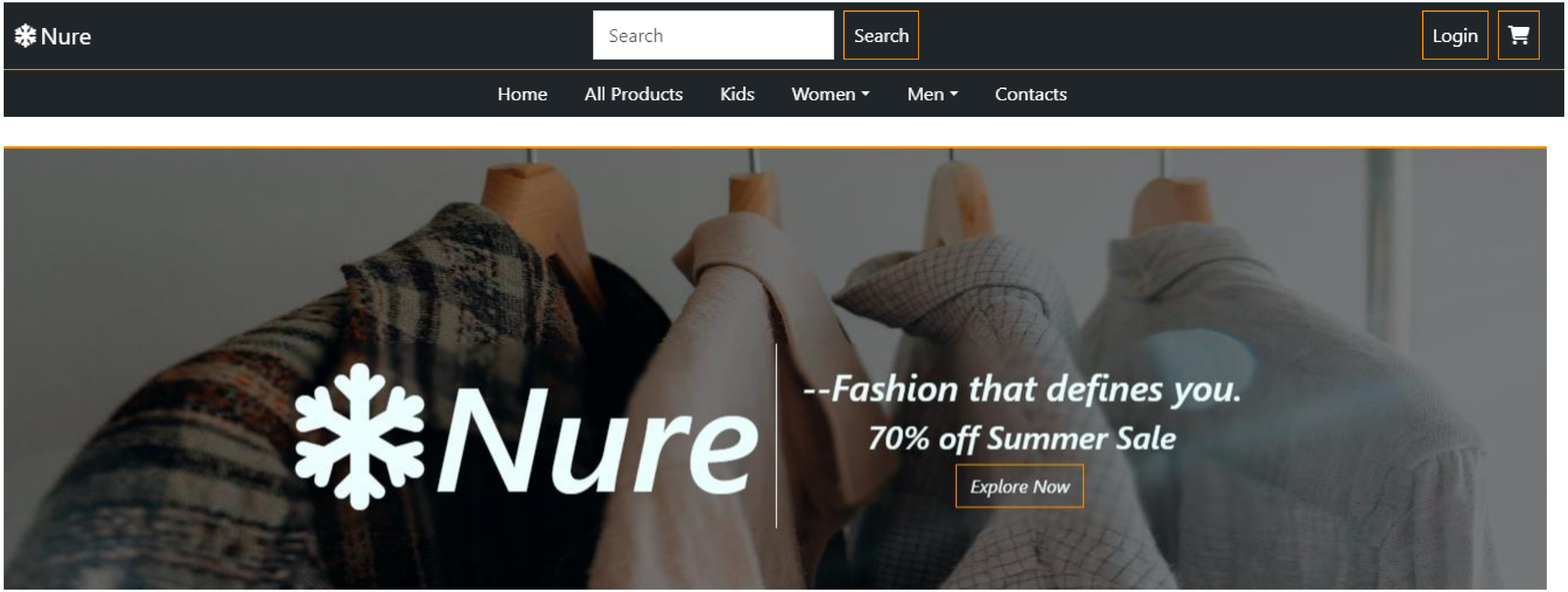
TEMPLATE

- A template is HTML that tells Angular how to render a component
- Templates include data bindings as well as other components and directives
- Angular 2 leverages native DOM events and properties which dramatically reduces the need for a ton of built- in directives
- Angular 2 leverages shadow DOM to do some really interesting things with view encapsulation

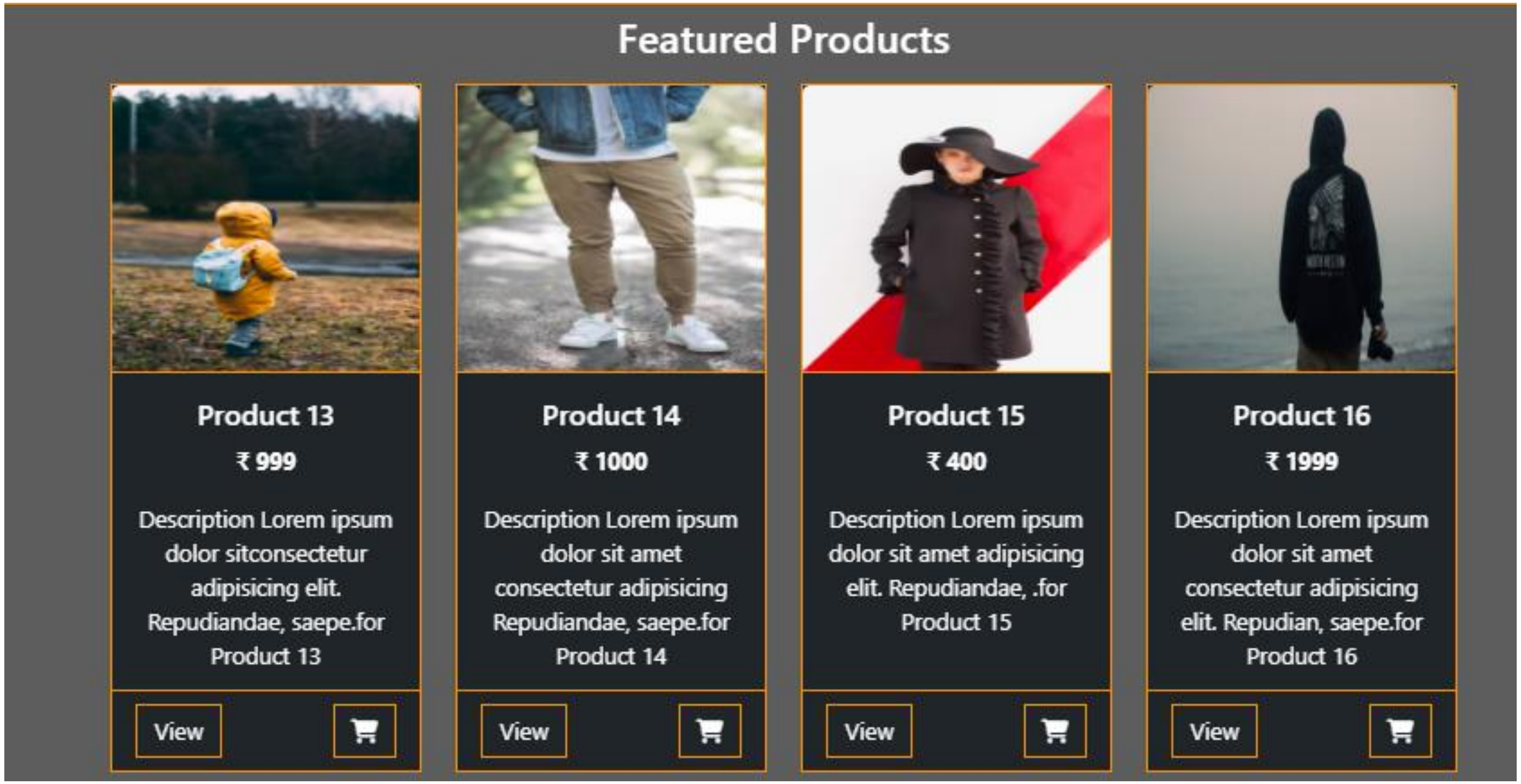
ACTIVITY

- Creating Home page for ecommerce portal using Angular.
    - **Header Component (include navbar)**
    - **Slider component**
    - **Featured products component**
    - **Footer Component**

SAMPLE OUTPUT HEADER & HOME COVER



FEATURED PRODUCTS



Featured Products

Product 13

₹ 999

Description Lorem ipsum dolor sitconsectetur adipisicing elit. Repudiandae, saepe.for Product 13

View

Product 14

₹ 1000

Description Lorem ipsum dolor sit amet consectetur adipisicing Repudiandae, saepe.for Product 14

View

Product 15

₹ 400

Description Lorem ipsum dolor sit amet adipisicing elit. Repudiandae, .for Product 15

View

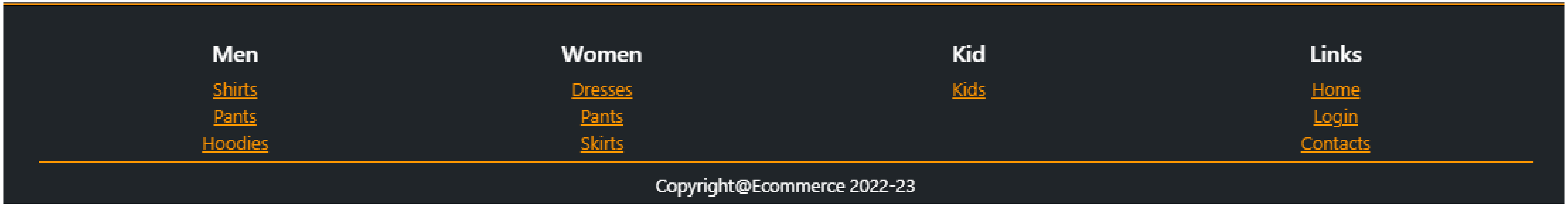
Product 16

₹ 1999

Description Lorem ipsum dolor sit amet consectetur adipisicing elit. Repudian, saepe.for Product 16

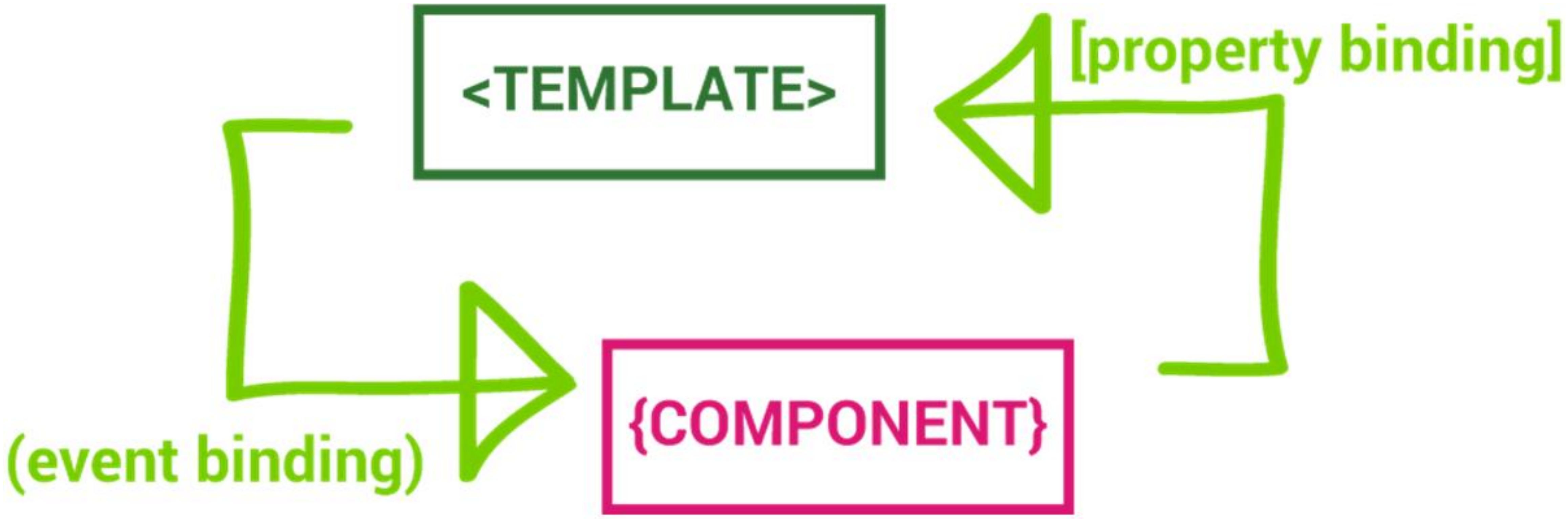
View

FOOTER



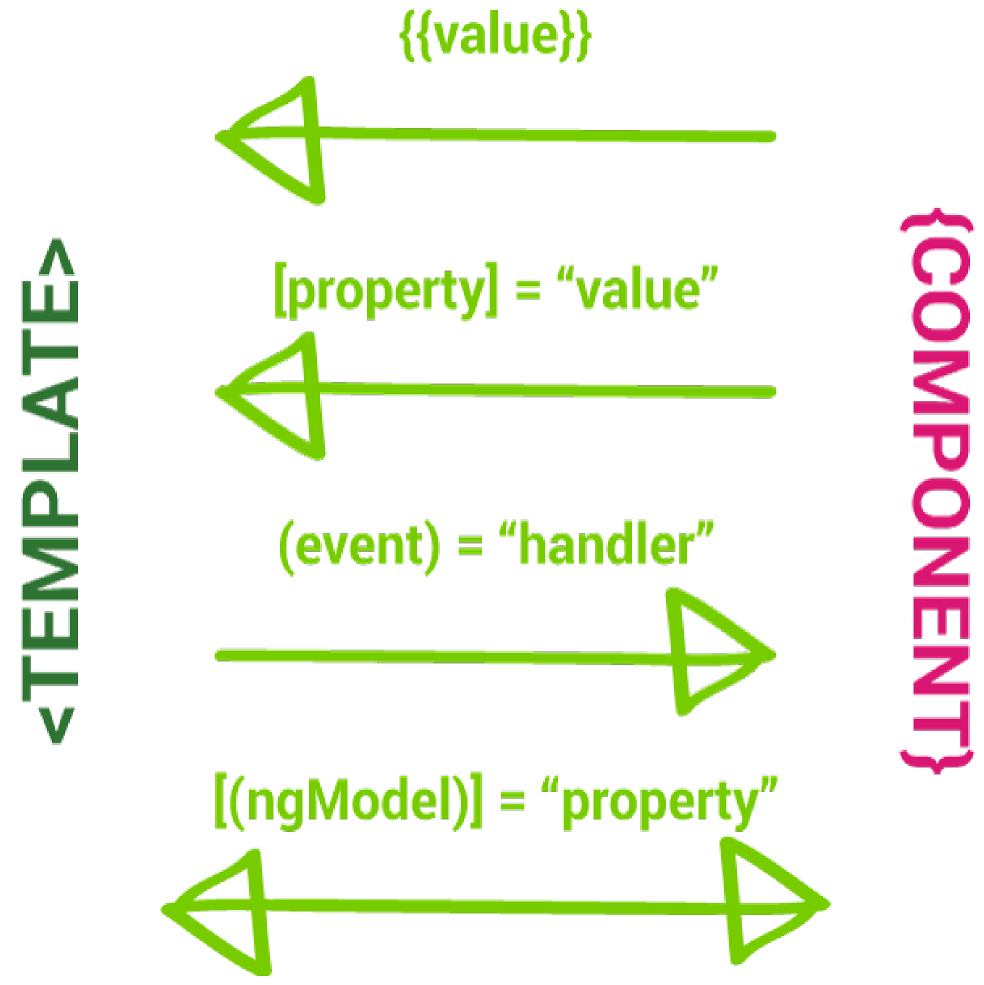
Men	Women	Kid	Links
Shirts	Dresses	Kids	Home
Pants	Pants		Login
Hoodies	Skirts		Contacts

Copyright@Ecommerce 2022-23


[property binding]
<TEMPLATE>
{COMPONENT}
(event binding)

DATA BINDING

- Enables data to flow from the component to template and vice-versa
- Includes interpolation, property binding, event binding, and two-way binding (property binding and event binding combined)
- The binding syntax has expanded but the result is a much smaller framework footprint


{{value}}
[property] = "value"
(event) = "handler"
[(ngModel)] = "property"
<TEMPLATE>
{COMPONENT}