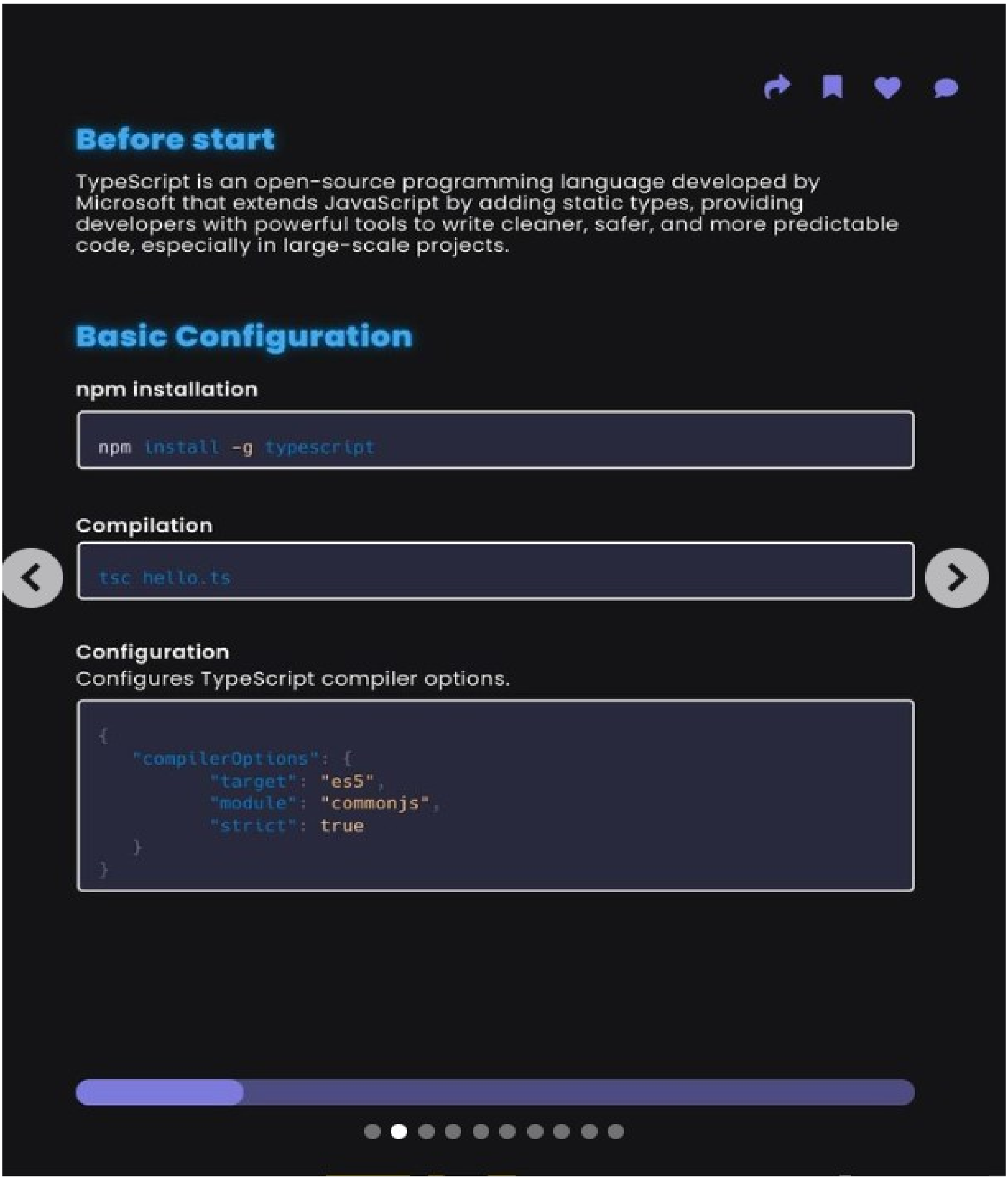
TYPESCRIPT CHEATSHEET



Before start

TypeScript is an open-source programming language developed by Microsoft that extends JavaScript by adding static types, providing developers with powerful tools to write cleaner, safer, and more predictable code, especially in large-scale projects.

Basic Configuration

npm installation

```
npm install -g typescript
```

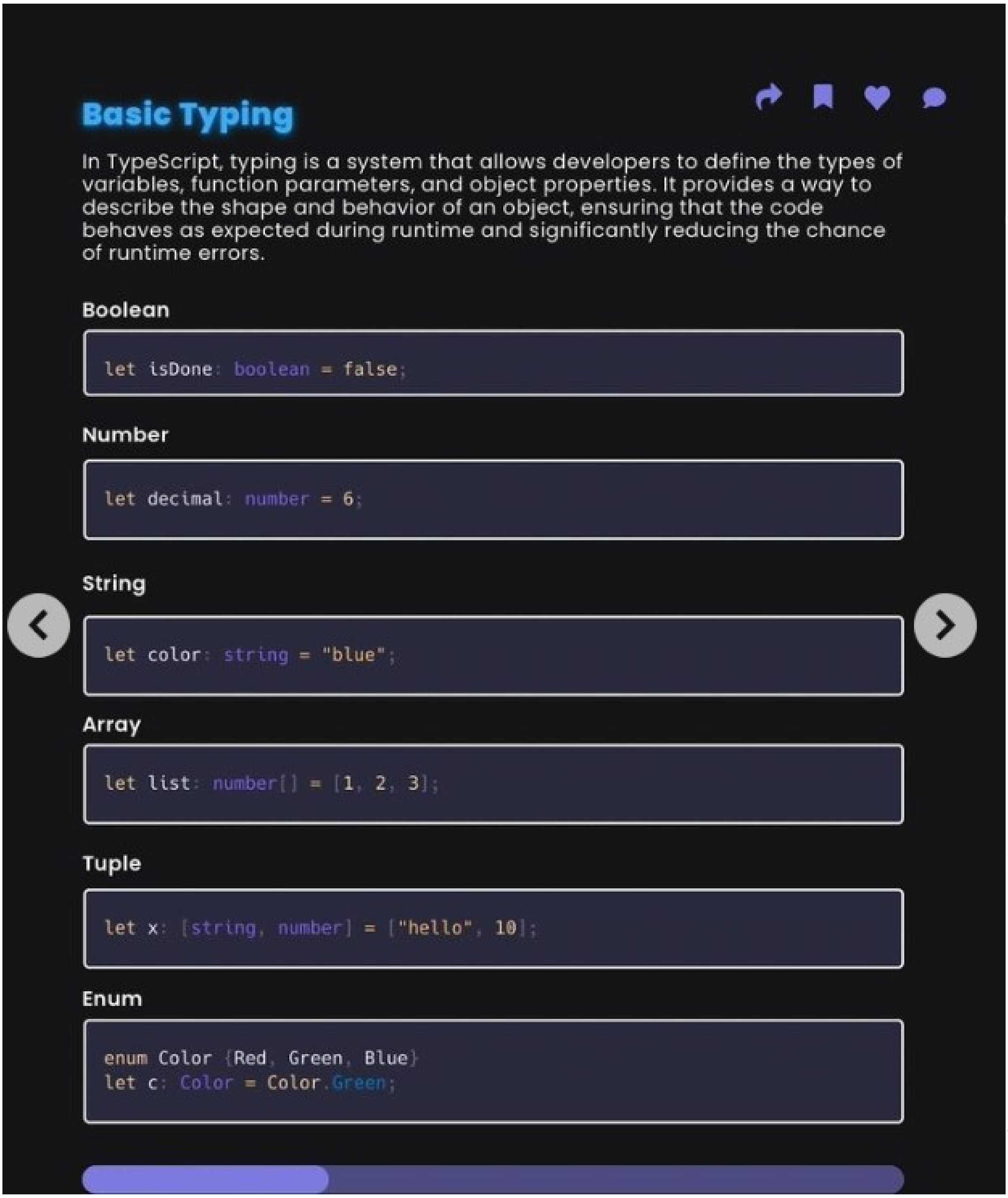
Compilation

```
tsc hello.ts
```

Configuration

Configures TypeScript compiler options.

```
{
   "compilerOptions": {
          "target": "es5",
          "module": "commonjs",
          "strict": true
   }
}
```



Basic Typing

In TypeScript, typing is a system that allows developers to define the types of variables, function parameters, and object properties. It provides a way to describe the shape and behavior of an object, ensuring that the code behaves as expected during runtime and significantly reducing the chance of runtime errors.

Boolean

```
let isDone: boolean = false;
```

Number

```
let decimal: number = 6;
```

String

```
let color: string = "blue";
```

Array

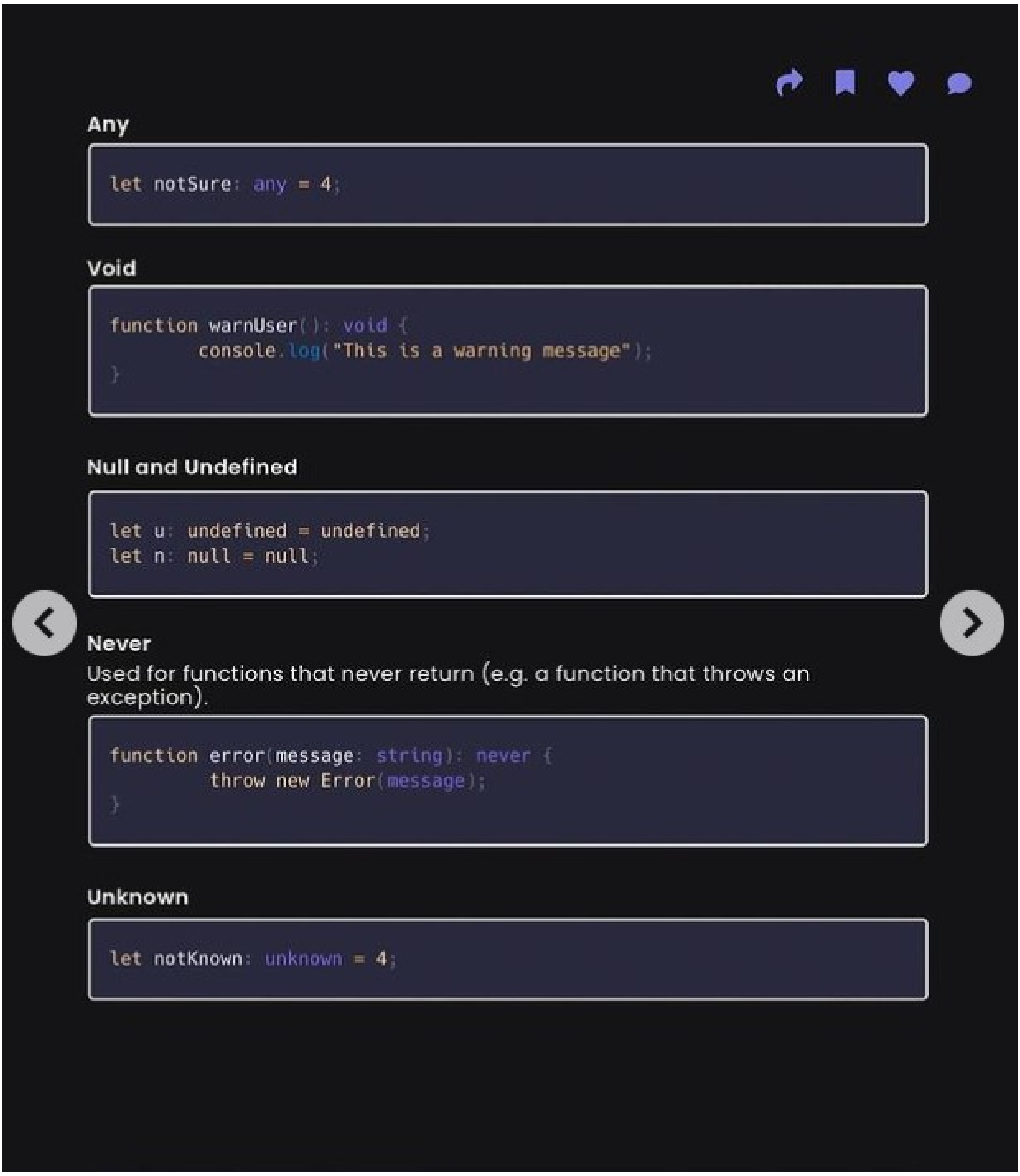
```
let list: number[] = [1, 2, 3];
```

Tuple

```
let x: [string, number] = ["hello", 10];
```

Enum

```
enum Color {Red, Green, Blue}
let c: Color = Color.Green;
```



Any

```
let notSure: any = 4;
```

Void

```
function warnUser(): void {
        console.log("This is a warning message");
}
```

Null and Undefined

```
let u: undefined = undefined;
let n: null = null;
```

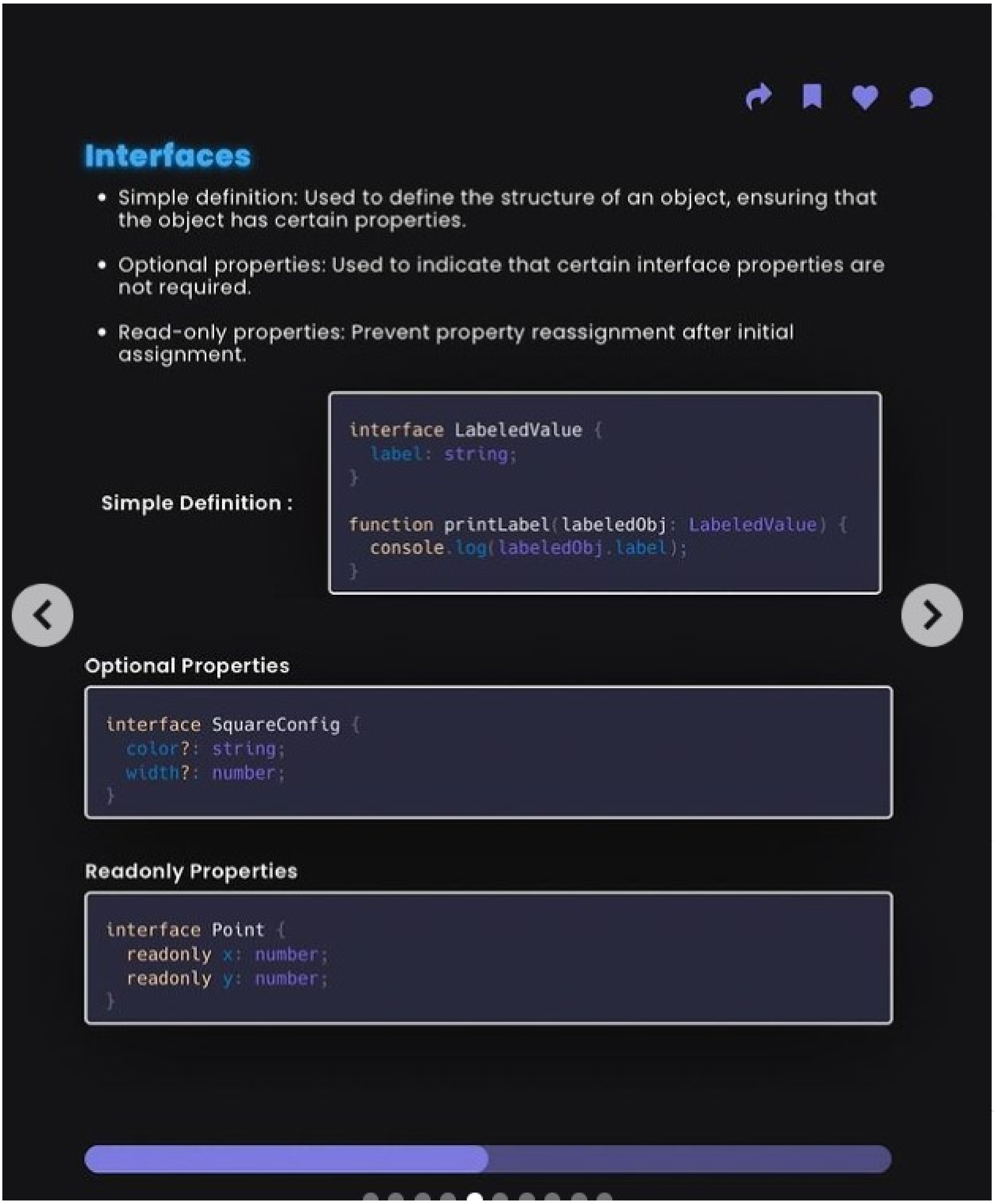
Never

Used for functions that never return (e.g. a function that throws an exception).

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

Unknown

```
let notKnown: unknown = 4;
```



Interfaces

- Simple definition: Used to define the structure of an object, ensuring that the object has certain properties.
- Optional properties: Used to indicate that certain interface properties are not required.
- Read-only properties: Prevent property reassignment after initial assignment.

Simple Definition :

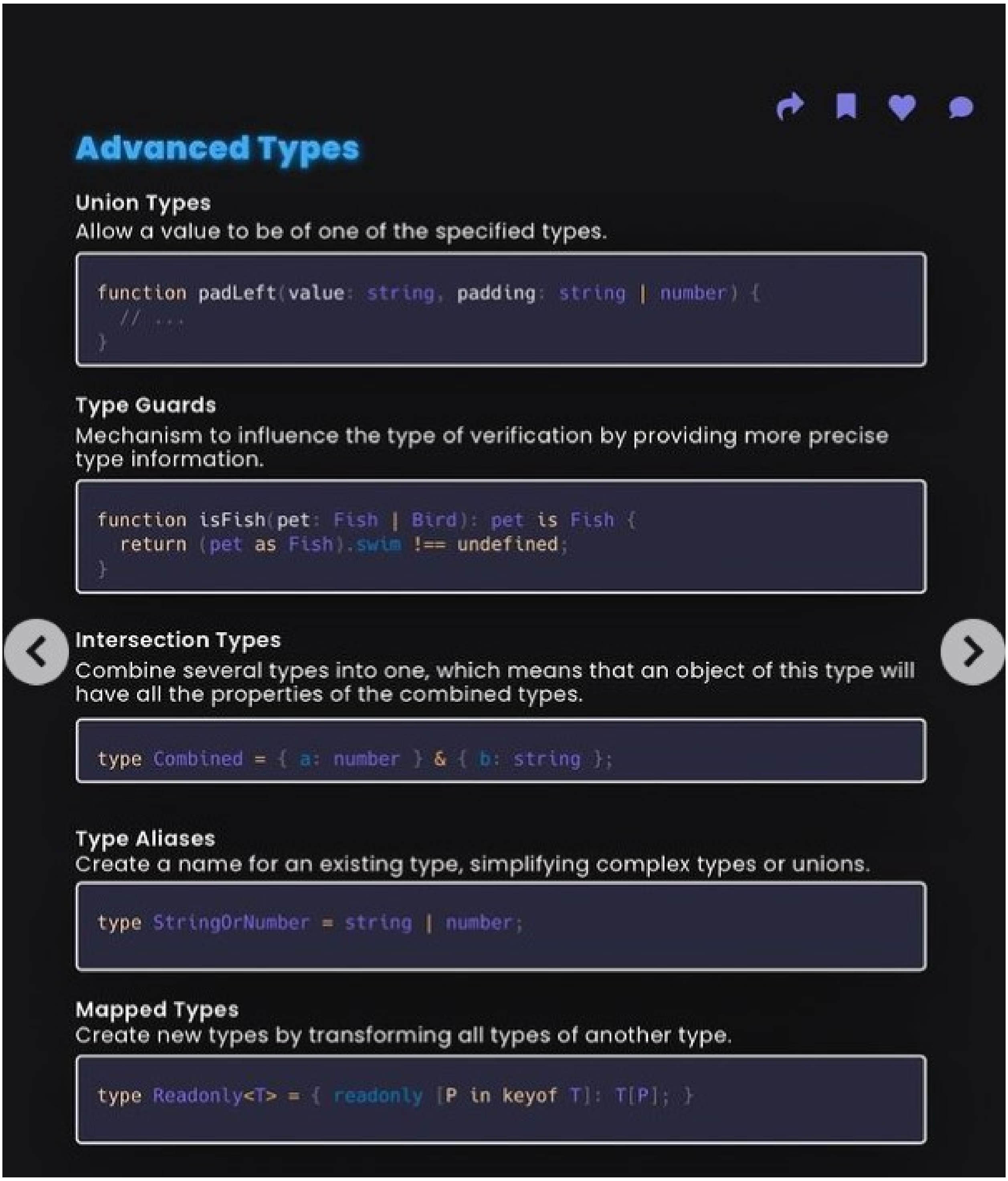
```
interface LabeledValue {
  label: string;
}

function printLabel(labeledObj: LabeledValue) {
  console.log(labeledObj.label);
}
```

Optional Properties

```
interface SquareConfig {
  color?: string;
  width?: number;
}
```

Readonly Properties

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



Advanced Types

Union Types
Allow a value to be of one of the specified types.

```
function padLeft(value: string, padding: string | number) {
  // ...
}
```

Type Guards
Mechanism to influence the type of verification by providing more precise type information.

```
function isFish(pet: Fish | Bird): pet is Fish {
  return (pet as Fish).swim !== undefined;
}
```

Intersection Types
Combine several types into one, which means that an object of this type will have all the properties of the combined types.

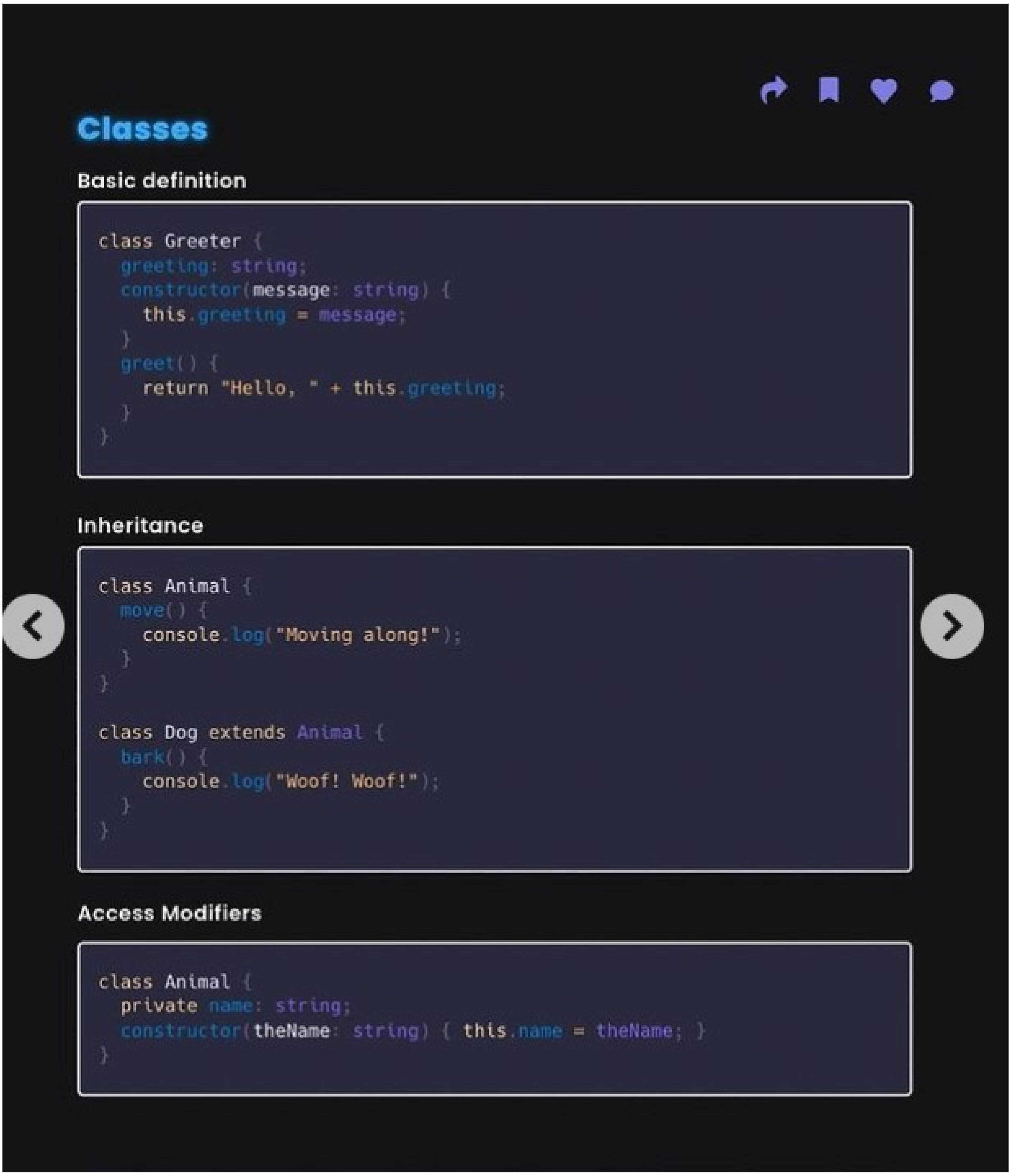
```
type Combined = { a: number } & { b: string };
```

Type Aliases
Create a name for an existing type, simplifying complex types or unions.

```
type StringOrNumber = string | number;
```

Mapped Types
Create new types by transforming all types of another type.

```
type Readonly<T> = { readonly [P in keyof T]: T[P]; }
```



Classes

Basic definition

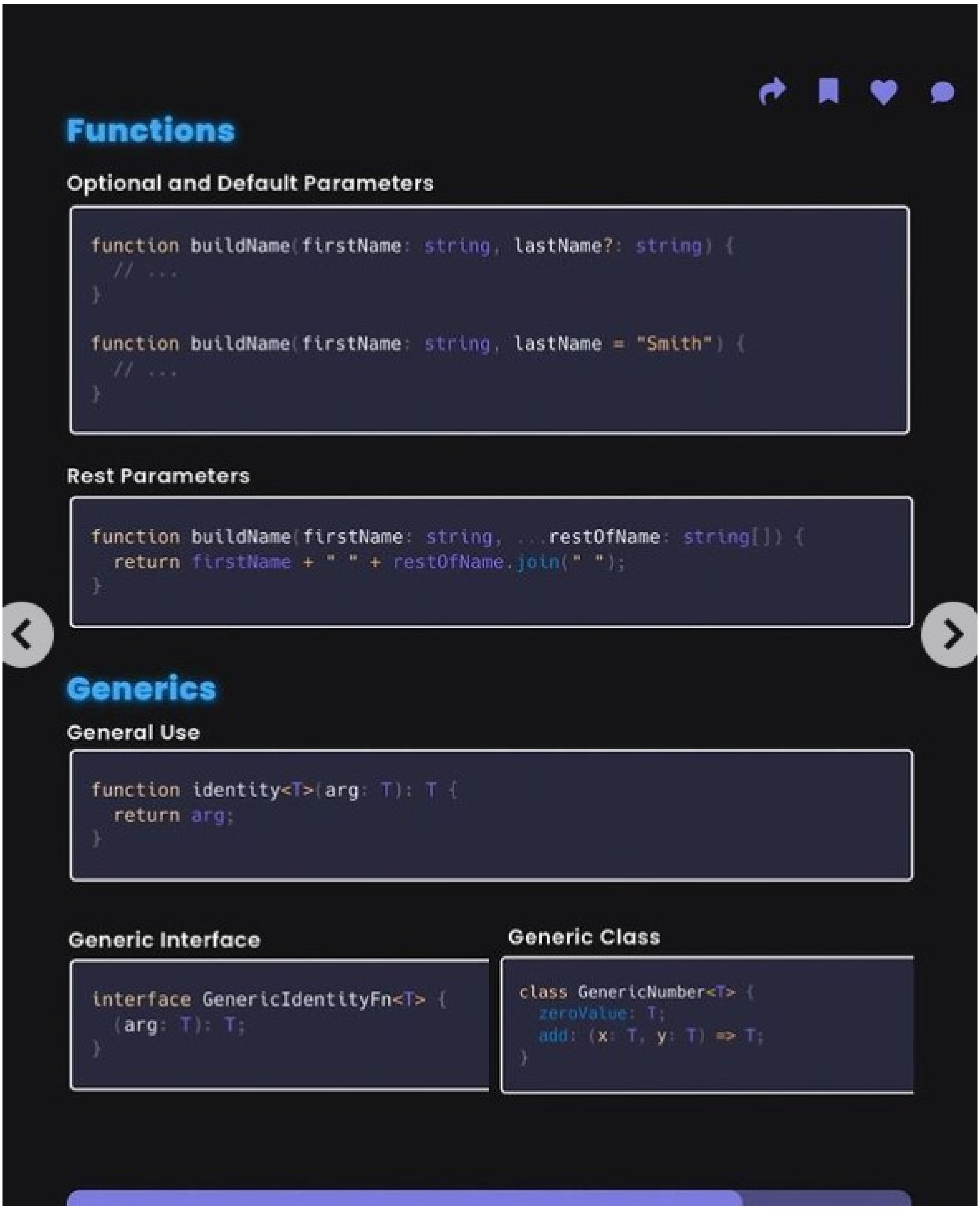
```
class Greeter {
  greeting: string;
  constructor(message: string) {
    this.greeting = message;
  }
  greet() {
    return "Hello, " + this.greeting;
  }
}
```

Inheritance

```
class Animal {
  move() {
    console.log("Moving along!");
  }
}

class Dog extends Animal {
  bark() {
    console.log("Woof! Woof!");
  }
}
```

Access Modifiers

```
class Animal {
  private name: string;
  constructor(theName: string) { this.name = theName; }
}
```



Functions

Optional and Default Parameters

```
function buildName(firstName: string, lastName?: string) {
  // ...
}

function buildName(firstName: string, lastName = "Smith") {
  // ...
}
```

Rest Parameters

```
function buildName(firstName: string, ...restOfName: string[]) {
  return firstName + " " + restOfName.join(" ");
}
```

Generics

General Use

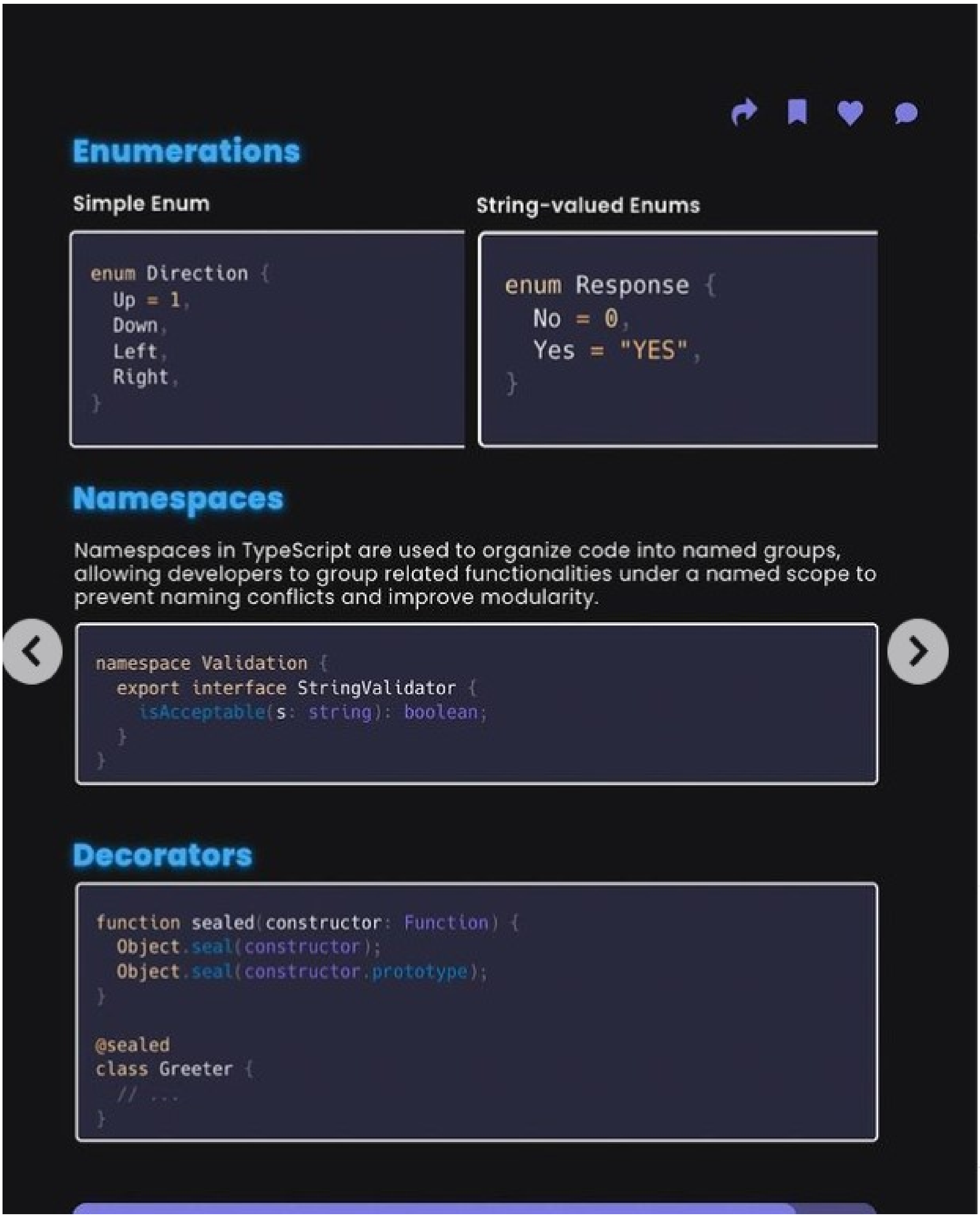
```
function identity<T>(arg: T): T {
  return arg;
}
```

Generic Interface

```
interface GenericIdentityFn<T> {
  (arg: T): T;
}
```

Generic Class

```
class GenericNumber<T> {
  zeroValue: T;
  add: (x: T, y: T) => T;
}
```



Enumerations

Simple Enum

```
enum Direction {
  Up = 1,
  Down,
  Left,
  Right,
}
```

String-valued Enums

```
enum Response {
  No = 0,
  Yes = "YES",
}
```

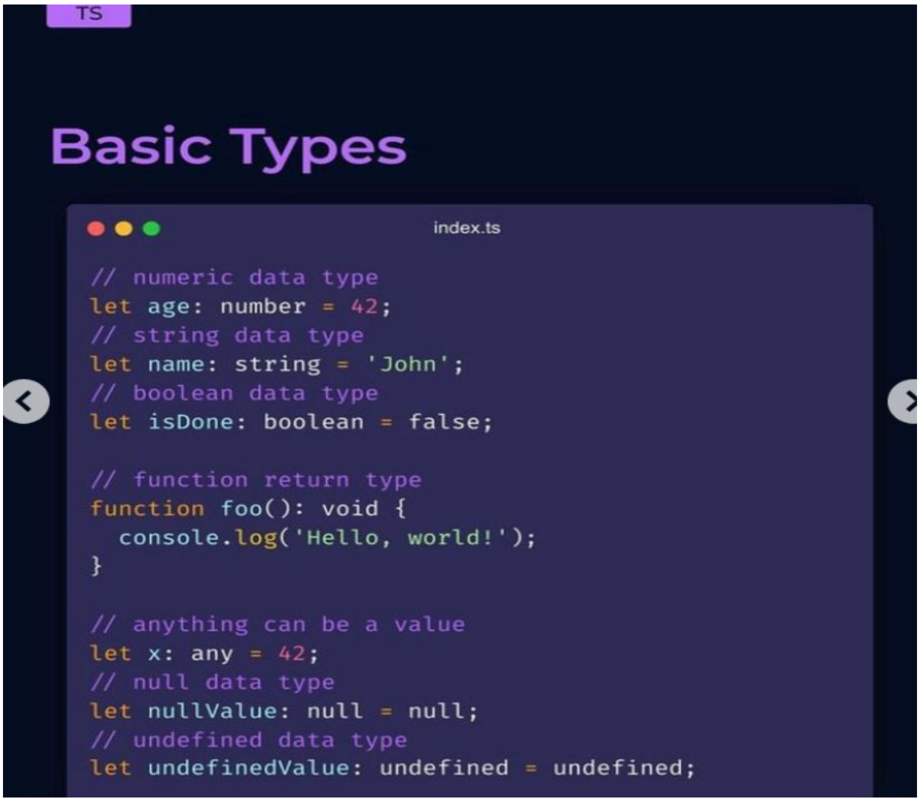
Namespaces

Namespaces in TypeScript are used to organize code into named groups, allowing developers to group related functionalities under a named scope to prevent naming conflicts and improve modularity.

```
namespace Validation {
  export interface StringValidator {
    isAcceptable(s: string): boolean;
  }
}
```

Decorators

```
function sealed(constructor: Function) {
  Object.seal(constructor);
  Object.seal(constructor.prototype);
}

@sealed
class Greeter {
  // ...
}
```

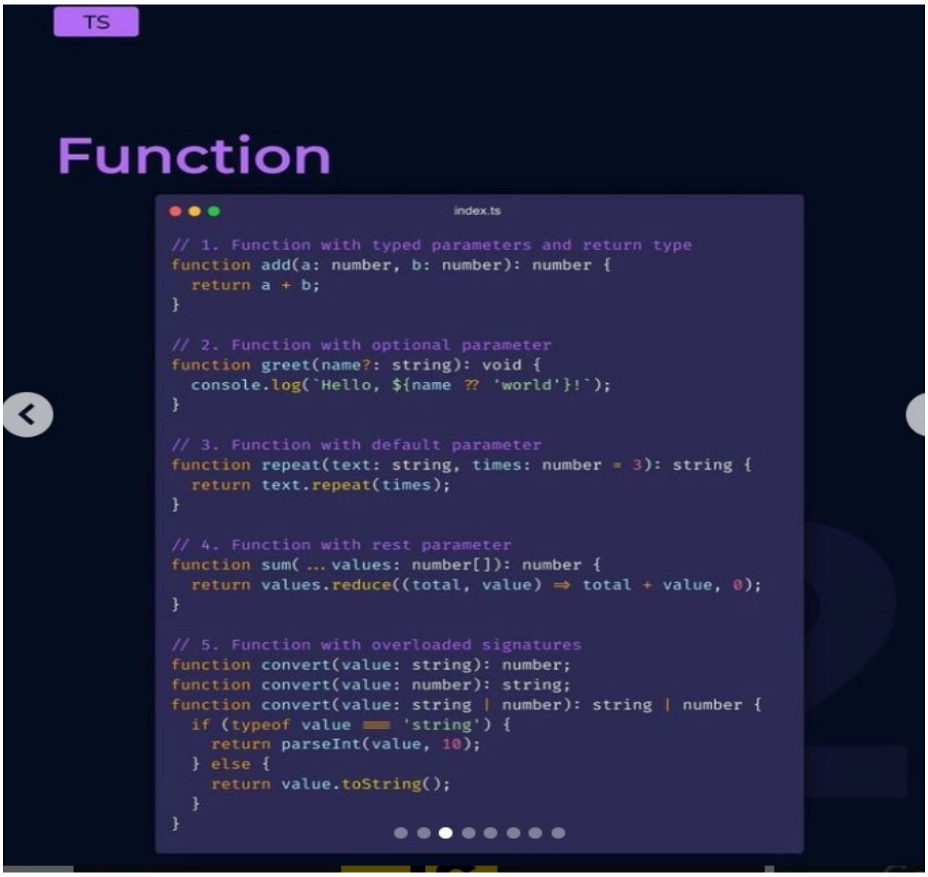


TS

Basic Types

index.ts

```ts
// numeric data type
let age: number = 42;
// string data type
let name: string = 'John';
// boolean data type
let isDone: boolean = false;

// function return type
function foo(): void {
  console.log('Hello, world!');
}

// anything can be a value
let x: any = 42;
// null data type
let nullValue: null = null;
// undefined data type
let undefinedValue: undefined = undefined;
```

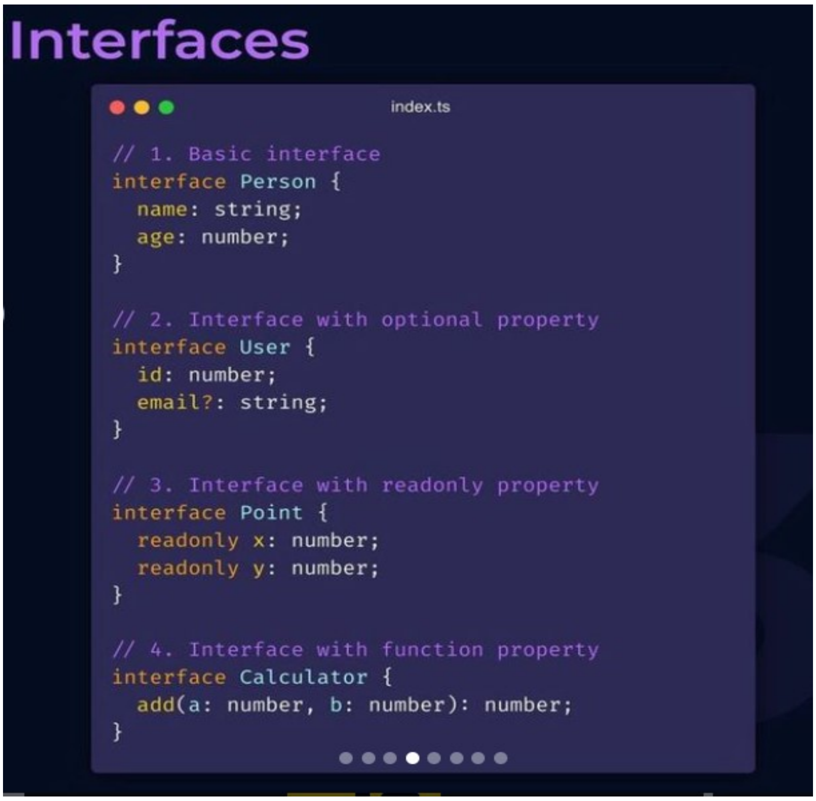


TS

Function

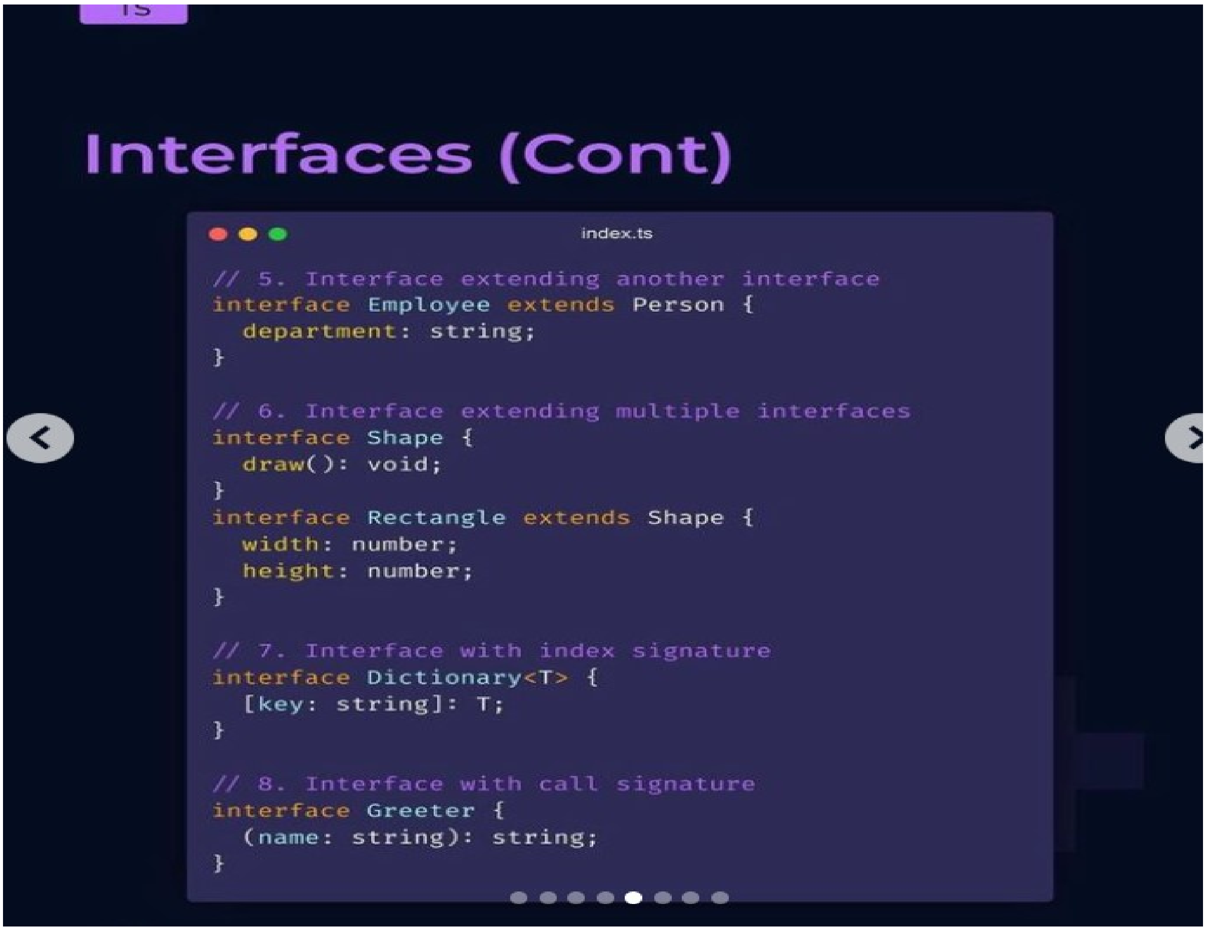
index.ts

```ts
// 1. Function with typed parameters and return type
function add(a: number, b: number): number {
  return a + b;
}

// 2. Function with optional parameter
function greet(name?: string): void {
  console.log(`Hello, ${name ?? 'world'}!`);
}

// 3. Function with default parameter
function repeat(text: string, times: number = 3): string {
  return text.repeat(times);
}

// 4. Function with rest parameter
function sum(...values: number[]): number {
  return values.reduce((total, value) => total + value, 0);
}

// 5. Function with overloaded signatures
function convert(value: string): number;
function convert(value: number): string;
function convert(value: string | number): string | number {
  if (typeof value === 'string') {
    return parseInt(value, 10);
  } else {
    return value.toString();
  }
}
```



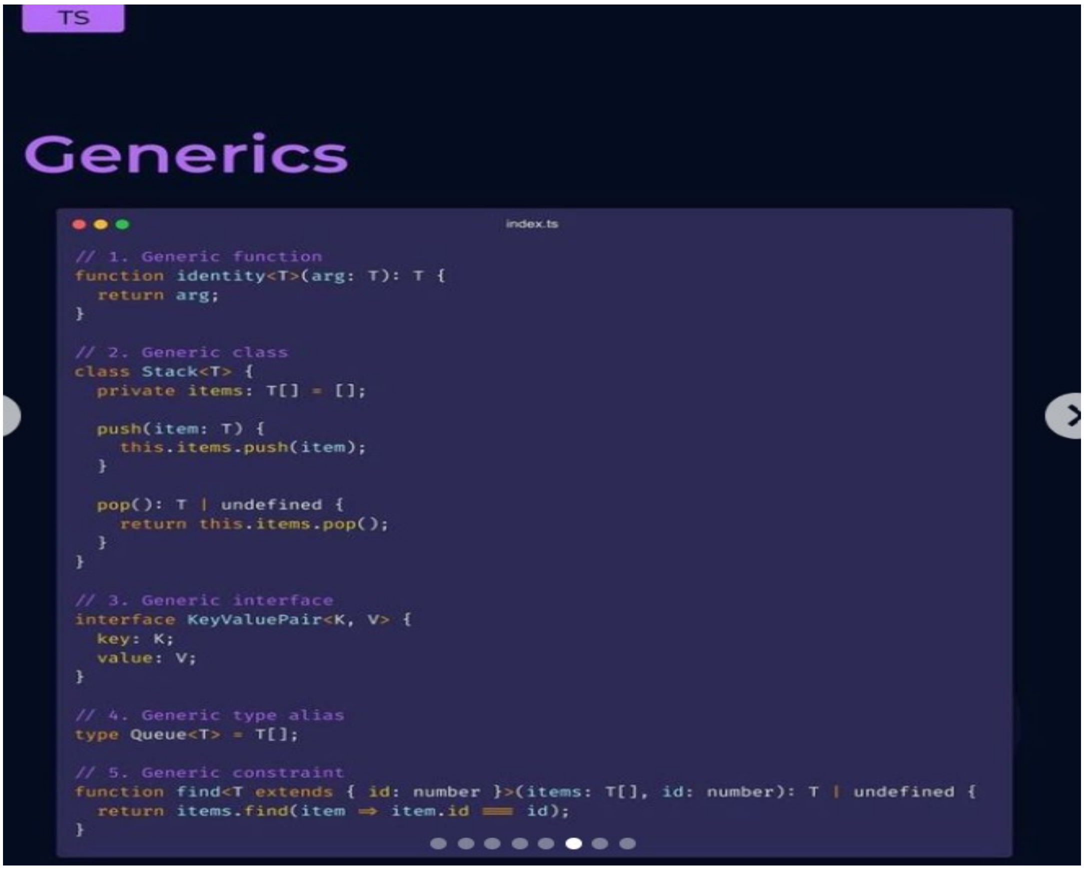
Interfaces

```
// 1. Basic interface
interface Person {
  name: string;
  age: number;
}

// 2. Interface with optional property
interface User {
  id: number;
  email?: string;
}

// 3. Interface with readonly property
interface Point {
  readonly x: number;
  readonly y: number;
}

// 4. Interface with function property
interface Calculator {
  add(a: number, b: number): number;
}
```



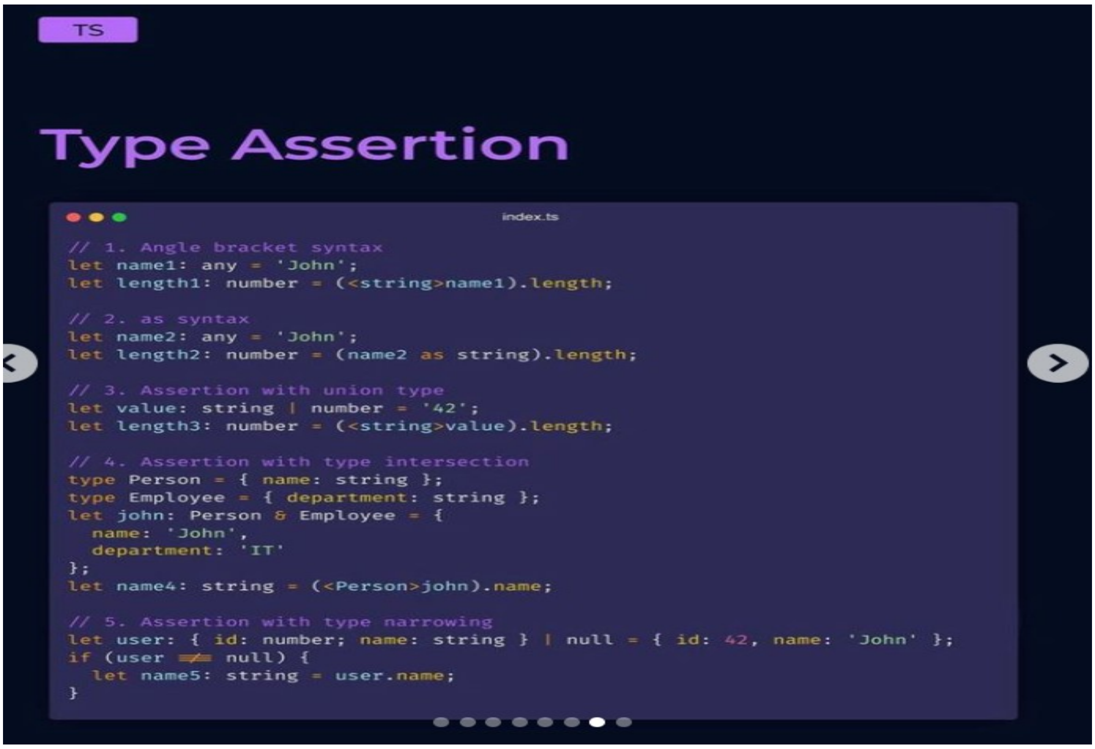
Interfaces (Cont)

```
// 5. Interface extending another interface
interface Employee extends Person {
  department: string;
}

// 6. Interface extending multiple interfaces
interface Shape {
  draw(): void;
}
interface Rectangle extends Shape {
  width: number;
  height: number;
}

// 7. Interface with index signature
interface Dictionary<T> {
  [key: string]: T;
}

// 8. Interface with call signature
interface Greeter {
  (name: string): string;
}
```



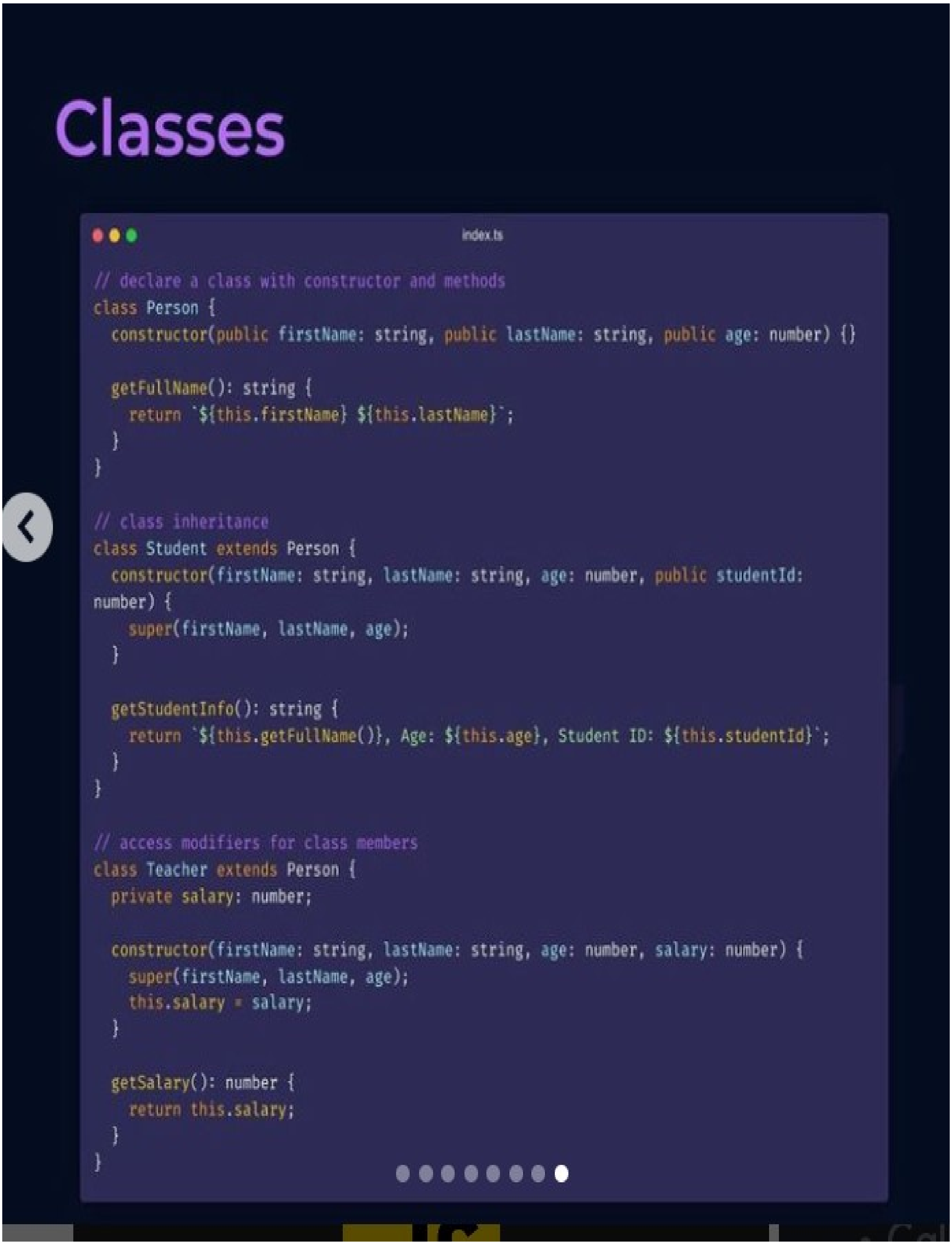
Generics

```typescript
// 1. Generic function
function identity<T>(arg: T): T {
  return arg;
}

// 2. Generic class
class Stack<T> {
  private items: T[] = [];

  push(item: T) {
    this.items.push(item);
  }

  pop(): T | undefined {
    return this.items.pop();
  }
}

// 3. Generic interface
interface KeyValuePair<K, V> {
  key: K;
  value: V;
}

// 4. Generic type alias
type Queue<T> = T[];

// 5. Generic constraint
function find<T extends { id: number }>(items: T[], id: number): T | undefined {
  return items.find(item => item.id === id);
}
```



Type Assertion

```typescript
// 1. Angle bracket syntax
let name1: any = 'John';
let length1: number = (<string>name1).length;

// 2. as syntax
let name2: any = 'John';
let length2: number = (name2 as string).length;

// 3. Assertion with union type
let value: string | number = '42';
let length3: number = (<string>value).length;

// 4. Assertion with type intersection
type Person = { name: string };
type Employee = { department: string };
let john: Person & Employee = {
  name: 'John',
  department: 'IT'
};
let name4: string = (<Person>john).name;

// 5. Assertion with type narrowing
let user: { id: number; name: string } | null = { id: 42, name: 'John' };
if (user !== null) {
  let name5: string = user.name;
}
```



Classes

index.ts

```
// declare a class with constructor and methods
class Person {
  constructor(public firstName: string, public lastName: string, public age: number) {}

  getFullName(): string {
    return `${this.firstName} ${this.lastName}`;
  }
}

// class inheritance
class Student extends Person {
  constructor(firstName: string, lastName: string, age: number, public studentId:
number) {
    super(firstName, lastName, age);
  }

  getStudentInfo(): string {
    return `${this.getFullName()}, Age: ${this.age}, Student ID: ${this.studentId}`;
  }
}

// access modifiers for class members
class Teacher extends Person {
  private salary: number;

  constructor(firstName: string, lastName: string, age: number, salary: number) {
    super(firstName, lastName, age);
    this.salary = salary;
  }

  getSalary(): number {
    return this.salary;
  }
}
```