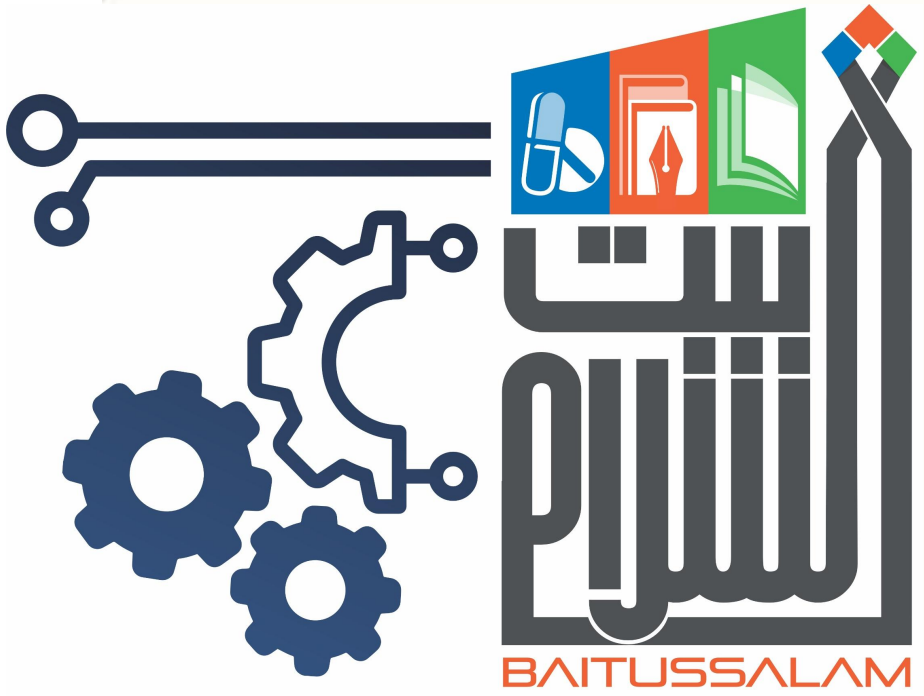


BAITUSSALAM
—TECH PARK—

Class Agenda

BAITUSSALAM
—TECH PARK—

Advanced Classes, Typescript, Getter Setter, and Singleton

Private Properties and Methods

BAITUSSALAM
—TECH PARK—

Private properties are class properties that cannot be accessed or modified from outside the class.

It serves to encapsulate the property, restricting its access to within the class itself.

Syntax: Private properties are declared using the # prefix.

In **Typescript** we write private keyword before the property name private balance: number;

```
class BankAccount {
  #balance

  constructor(initialBalance) {
    this.#balance = initialBalance
  }

  getBalance() {
    return this.#balance
  }

  deposit(amount) {
    this.#balance += amount
  }
}

const account = new BankAccount(100)
account.deposit(500)

console.log(account.getBalance()) // 600

console.log(account.#balance)
// SyntaxError: Private field '#balance'
// must be declared in an enclosing class
```

Protected Properties and Methods

Protected properties and methods are members of a class that are **accessible within the class itself and by instances of subclasses**, but **not accessible from outside the class hierarchy.** This means that they can be used to share functionality between a class and its subclasses while still restricting direct access from other parts of the program.

Syntax: Protected properties are declared using the _ prefix in JavaScript.

In Typescript we write **protected** keyword before the property name

protected balance: number;

Protected Properties and Methods

BAITUSSALAM
—TECH PARK—

Typescript Examples

```typescript
class Vehicle {
  protected speed: number

  constructor(speed: number) {
    this.speed = speed
  }

  protected accelerate(increment: number): void {
    this.speed += increment
  }
}
```

```typescript
class Car extends Vehicle {
  private brand: string

  constructor(speed: number, brand: string) {
    super(speed)
    this.brand = brand
  }

  public boost(): void {
    this.accelerate(20)
    console.log(`Boosted speed: ${this.speed}`)
  }
}

const car = new Car(100, 'Toyota')
car.boost() // "Boosted speed: 120"
console.log(car.speed)
// Error: Property 'speed' is protected and only accessible
// within class 'Vehicle' and its subclasses.
console.log(car.accelerate(20))
// Error: Property 'accelerate' is protected and only accessible
//within class 'Vehicle' and its subclasses.
```

Abstract Classes

An abstract class is a class that cannot be instantiated.

This means that you cannot use an abstract class to directly create an object.

```
abstract class Shape {
  name: string
  constructor(name: string) {
    this.name = name
  }
}

class Circle extends Shape {
  radius: number
  constructor(name: string, radius: number) {
    super(name)
    this.radius = radius
  }
}

const myShape = new Shape('My shape')
// This will throw an Error

const shortCircle = new Circle('Short Circle', 0.5)
// This will work fine.
```

Static Methods

Static methods are methods defined on a class that can be called **without creating an instance of the class**. They are typically used for **utility functions** that are related to the class but don't operate on instances of the class.

```
class MathUtils {
  static add(a, b) {
    return a + b
  }

  static multiply(a, b) {
    return a * b
  }
}

console.log(MathUtils.add(5, 3)) // 8
console.log(MathUtils.multiply(5, 3)) // 15

const mathUtils = new MathUtils()
console.log(mathUtils.add(5, 3))
// TypeError: mathUtils.add is not a function
```

Singleton Design Pattern

The **Singleton Pattern** ensures that a class has **only one instance** and provides a **global point of access** to it.

In short, single instance of an object throughout the application

```
class Authentication {
  static isUserLoggedIn

  constructor() {
    if (Authentication.instance) {
      return Authentication.instance
    }
    Authentication.instance = this
    this.isUserLoggedIn = false
  }

  loginUser() {
    this.isUserLoggedIn = true
    return 'User logged in'
  }

  logoutUser() {
    this.isUserLoggedIn = false
    return 'User logged out'
  }
}

const instance1 = new Authentication()
const instance2 = new Authentication()

console.log('Authentication instances ==>', instance1 === instance2)

console.log('login user', instance1.loginUser())
console.log('instance1', instance1)
```

Read-only properties (objects)

Object.freeze() method

The Object.freeze() static method freezes an object.

```
const person = {
  name: 'Ali',
  designation: 'Designer',
}

Object.freeze(person)

person.name = 'Fahad'
// Throws an error in strict mode

console.log(person.name)
// Expected output: Ali
```

Object.defineProperty()

Object.defineProperty() allows you to define a property with specific attributes.

Setting the writable attribute to false makes the property read-only.

```
const students = {}

Object.defineProperty(obj, 'name', {
  value: 'Alice',
  writable: false,
  configurable: false,
})

obj.name = 'Bob' // This will have no effect
delete obj.name // This will also have no effect

console.log(obj.name) // 'Alice'
```

Object.seal()

Object.seal() prevents adding or removing properties from an object. However, it does not prevent modifying existing properties.

Getter and Setter

In JavaScript, there are two kinds of object properties:

1. Data properties
2. Accessor properties

JavaScript Getter
getter methods are used to access the properties of an object.

JavaScript Setter
setter methods are used to change or mutate the values of an object.

```
104  const product = {
105    title: 'Laptop',
106    price: 40000,
107    get details() {
108      return `${product.title} current price is ${product.price}`
109    },
110    set details(value) {
111      const parts = value.split(' ')
112      this.title = parts[0]
113      this.price = parts[parts.length - 1]
114    },
115  }
116
117  product.details = 'headphone price is 2000'
118
119  console.log('product', product)
120  console.log('details', product.details)
121
```

JS Object Dynamic Property

Dynamic key properties in JavaScript objects allow you to use variables as property keys.

Syntax: Enclose the variable in square brackets [] when defining the property.

Use Case: Useful for creating properties based on variable values at runtime.

```
130  let key = 'dynamicKey'
131  let value = 'This is a dynamic value'
132
133  let obj = {
134      [key]: value,
135  }
136
137  console.log(obj.dynamicKey)
138  // Output: This is a dynamic value
139
```

Object Dynamic Property Exercise

Building an Object from User Input.

```
let userInputs = [
    { key: 'username', value: 'johndoe' },
    { key: 'email', value: 'john@example.com' },
    { key: 'password', value: '12345' }
]
```

Loop over the userInputs the output will be single object like this

```
Output: {
    username: 'johndoe',
    email: 'john@example.com',
    password: '12345'
}
```

Typescript Compiler

Strict Mode: compile it after each save
Watch Mode: compile continuously **--watch**

Generate **tsconfig.json** file using **tsc –init**

Compile the ts file to js
tsc filename.ts

Compile typescript in desired directory

Oraganize typescript project
rootDir: ./src => ts source code
outDir: ./dist => compile to ts code this folder

Typescript Enums

Enums allow us to define or declare a collection of related values that can be numbers or strings as a set of named constants. Unlike some types available in TypeScript

```typescript
enum UserRole {
  Admin = 'ADMIN',
  User = 'USER',
  Guest = 'GUEST',
}

function checkAccess(role: UserRole) {
  if (role === UserRole.Admin) {
    console.log('Access granted to admin.')
  } else if (role === UserRole.User) {
    console.log('Access granted to user.')
  } else {
    console.log('Access denied.')
  }
}

checkAccess(UserRole.Admin) // Output: Access granted to admin.
checkAccess(UserRole.Guest) // Output: Access denied.
```

Typescript Enums Exercise

1. Create an Enum for Task Status
Define an enum named **TaskState** with three possible values:

1. ToDo with the string value 'TODO'
2. InProgress with the string value 'IN_PROGRESS'
3. Done with the string value 'DONE'

Call **updateTaskStatus** with each enum value to verify the function works correctly.

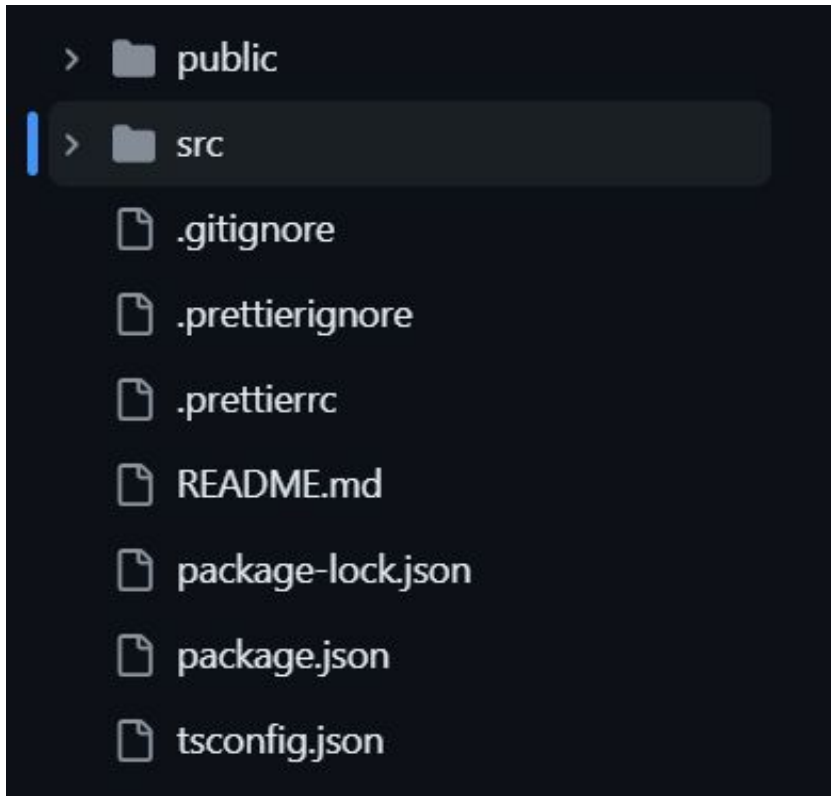
2. Create the Function **updateTaskStatus**

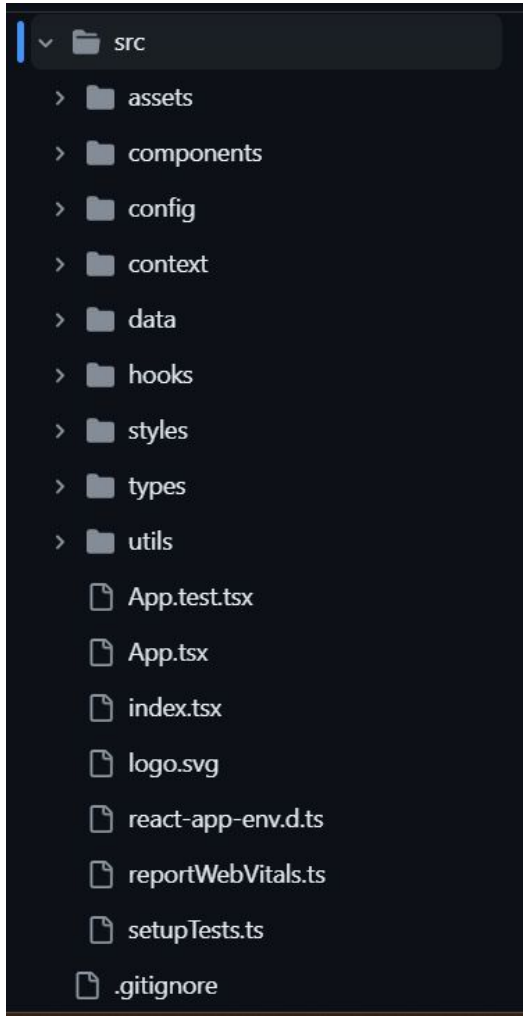
Define a function named updateTaskStatus that takes a parameter **status** of type **TaskState**. Inside the function, use a switch statement to print a message based on the status value.

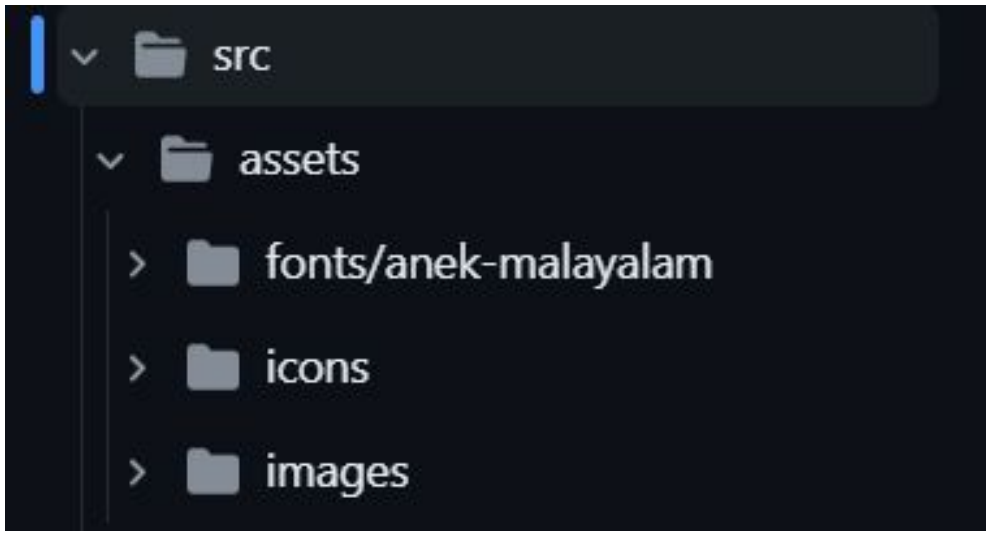
Handle Each Status in the Switch Statement

1. If status is TaskState.ToDo, print Task is in To Do.
2. If status is TaskState.InProgress, print Task is in Progress.
3. If status is TaskState.Done, print Task is Done.
Test the Function

Folder Structure for Scalable JS App







The End

BAITUSSALAM
—TECH PARK—