Project 24 Typescript

TypeScript

1. TypeScript Ortamının Hazırlanması

TypeScript kullanmaya başlamak için öncelikle TypeScript'i kurmanız gerekir. Bu, npm (Node Package Manager) kullanılarak yapılabilir:

```
npm install -g typescript
```

TypeScript dosyalarını `.ts` uzantısıyla oluşturabilirsiniz ve bu dosyaları JavaScript'e dönüştürmek için aşağıdaki komutu kullanabilirsiniz:

```
tsc dosya_adı.ts
```

2. Data Annotation

Data annotation, veri tiplerini belirlemek için kullanılan açıklamalardır. TypeScript'te bu, veri tiplerini belirtmek için kullanılır. Örneğin:

```
let isim: string = "Ahmet";
let yas: number = 30;
let aktif: boolean = true;
```

3. Data Type - Number, String, Boolean

TypeScript'te temel veri türleri şunlardır:

- **`number`**: Sayısal değerler için kullanılır.
- **`string`**: Metin değerleri için kullanılır.
- **`boolean`**: `true` veya `false` değerlerini temsil eder.

Örnek:

```
let age: number = 25;
let name: string = "John";
let isActive: boolean = true;
```

4. Data Type - Array

Diziler, aynı türdeki birden fazla değeri saklar. TypeScript'te diziler şu şekilde tanımlanır:

```
let numbers: number[] = [1, 2, 3, 4, 5];
let names: string[] = ["Alice", "Bob", "Charlie"];
```

5. Data Type - Tuple

Tuple, farklı türlerdeki birden fazla değeri belirli bir sırayla saklamak için kullanılır.

```
let person: [string, number] = ["John", 30];
```

6. Data Type - Object

Nesneler, birden fazla özelliği saklamak için kullanılır:

```
let person: { name: string; age: number } = { name: "Alice", age: 25 };
```

7. Data Type - Enum

Enum, belirli bir grup isimlendirilmiş sabit değerleri temsil eder.

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

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

8. Data Type - Union

Union, bir değişkenin birden fazla türde olabileceğini belirtir.

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

```
value = 42;
```

9. Data Type - Any

`any` türü, herhangi bir türdeki değeri kabul eder ve TypeScript'in tür kontrolünden kaçınmanıza olanak sağlar.

```
let data: any = "Hello";
data = 42;
data = true;
```

10. Data Type - Void

`void`, bir fonksiyonun hiçbir değer döndürmeyeceğini belirtir.

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

11. Data Type - Never

`never`, bir fonksiyonun hiçbir zaman başarıyla tamamlanmayacağını belirtir (örneğin, sürekli bir döngü veya hata fırlatma).

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

12. Type - Inference

TypeScript, değişkenlerin türlerini otomatik olarak çıkarabilir. Bu, tür belirtmek zorunda kalmadan TypeScript'in değişken türünü belirlemesini sağlar.

```
let message = "Hello"; // TypeScript, message değişkeninin türünü string olarak belirler.
```

13. Type - Assertion

Tür dönüşümü (type assertion), TypeScript'in tür çıkarımını zorla belirtmek için kullanılır.

```
let someValue: any = "This is a string";
let strLength: number = (someValue as string).length;
```

14. Type - If Else ve Ternary Operator

If-else:

```
let age: number = 18;

if (age >= 18) {
  console.log("Adult");
} else {
  console.log("Not Adult");
}
```

Ternary Operator:

```
let age: number = 18;
let status: string = age >= 18 ? "Adult" : "Not Adult";
```

15. Switch Case

`switch` ifadesi, bir değere göre birden fazla durumu kontrol etmek için kullanılır.

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

switch (color) {
  case "red":
    console.log("Color is red");
    break;
  case "blue":
    console.log("Color is blue");
    break;
  default:
    console.log("Color is unknown");
}
```

16. For Loop

`for` döngüsü, belirli bir sayıda tekrar etmek için kullanılır.

```
for (let i = 0; i < 5; i++) {
  console.log(i);
}
```

17. While Loop, Do While Loop ve Break

While:

```
let i = 0;
while (i < 5) {
  console.log(i);
  i++;
}
```

Do While:

```
let i = 0;
do {
  console.log(i);
  i++;
} while (i < 5);
```

Break:

```
for (let i = 0; i < 10; i++) {
  if (i === 5) {
    break;
  }
  console.log(i);
}
```

18. Function

Fonksiyonlar, tekrar kullanılabilir kod blokları oluşturur.

```
function greet(name: string): string {
  return `Hello, ${name}!`;
```

```
}
```

19. Optional Parameters ve Arrow Functions

Optional Parameters:

```
function greet(name: string, age?: number): string {
  if (age) {
    return `Hello, ${name}. You are ${age} years old.`;
  }
  return `Hello, ${name}`;
}
```

Arrow Functions:

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

20. Function Overloading

Fonksiyon aşırı yüklemesi, aynı isimle birden fazla fonksiyon tanımlamayı sağlar.

```
function greet(name: string): string;
function greet(name: string, age: number): string;
function greet(name: string, age?: number): string {
  if (age) {
    return `Hello, ${name}. You are ${age} years old.`;
  }
  return `Hello, ${name}`;
}
```

21. Rest Parameters

Rest parametreler, bir fonksiyona değişken sayıda argüman geçirmeyi sağlar.

```
function sum(...numbers: number[]): number {
  return numbers.reduce((total, num) => total + num, 0);
}
```

22. Class

Sınıflar, nesne tabanlı programlamada kullanılan temel yapı taşlarıdır.

```
class Person {
  name: string;
  constructor(name: string) {
    this.name = name;
  }
}
```

23. Access Modifiers (Public, Private, Protected)

- **Public:** Her yerden erişilebilir.
- **Private:** Sadece sınıf içinden erişilebilir.
- **Protected:** Sadece sınıf içinden ve sınıftan türemiş sınıflardan erişilebilir.

```
class Person {
  public name: string;
  private age: number;
  protected id: number;

  constructor(name: string, age: number, id: number) {
    this.name = name;
    this.age = age;
    this.id = id;
  }
}
```

24. Readonly

`readonly` , bir özelliğin sadece okunabilir olmasını sağlar, yani değeri sadece yapıcı (constructor) içinde ayarlanabilir ve daha sonra değiştirilemez.

```
class Person {
  readonly id: number;

  constructor(id: number) {
    this.id = id;
```

```
  }
}
```

25. Inheritance

Kalıtım, bir sınıfın başka bir sınıftan türemesini sağlar.

```
class Employee extends Person {
  position: string;

  constructor(name: string, age: number, id: number, position: string) {
    super(name, age, id);
    this.position = position;
  }
}
```

26. Static Methods ve Properties

Static yöntemler ve özellikler, sınıfın kendisine ait olup, sınıf örneklerine değil, doğrudan sınıfa aittir.

```
class MathUtil {
  static PI: number = 3.14;

  static areaOfCircle(radius: number): number {
    return MathUtil.PI * radius * radius;
  }
}
```

27. Abstract Class

Soyut sınıflar, diğer sınıflar tarafından genişletilmek üzere tasarlanır ve kendileri doğrudan örneklenemez.

```
abstract class Shape {
  abstract area(): number;
}
```

28. Interface Nedir ve Nas

ıl Kullanılır

Arayüzler, sınıfların bir yapı (veya sözleşme) sağlamasını garanti eder.

```
interface Shape {
  area(): number;
}

class Circle implements Shape {
  radius: number;
  constructor(radius: number) {
    this.radius = radius;
  }

  area(): number {
    return Math.PI * this.radius * this.radius;
  }
}
```

29. Interface Optional Parameters ve Readonly Function Type

Optional Parameters:

```
interface Config {
  width?: number;
  height?: number;
}
```

Readonly Function Type:

```
interface ReadonlyFunction {
  (param: string): string;
}

const greet: ReadonlyFunction = (param: string) => `Hello, ${param}`;
```

30. Interface Extend Etme ve Bir Class'a Interface Implement Etme

Extend Etme:

```ts
interface Person {
  name: string;
}

interface Employee extends Person {
  position: string;
}
```

Implement Etme:

```ts
class Worker implements Employee {
  name: string;
  position: string;

  constructor(name: string, position: string) {
    this.name = name;
    this.position = position;
  }
}
```

31. Type Intersection

Intersection (kesişim) türleri, birden fazla türü birleştirir.

```ts
type BusinessPartner = Person & { credit: number };
```

32. Type Guard

Type guard'lar, bir değerin belirli bir türde olup olmadığını kontrol eder.

```ts
function isString(value: any): value is string {
  return typeof value === 'string';
}

let value: any = "Hello";
if (isString(value)) {
  console.log(value.toUpperCase());
}
```

33. Generics

Jenerik türler, kodun tür bağımsız olarak çalışmasını sağlar.

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

34. Generic Constraints

Jenerik kısıtlamalar, jenerik türün belirli bir türde olması gerektiğini belirtir.

```
function logLength<T extends { length: number }>(item: T): void {
  console.log(item.length);
}
```

35. Interface'lerde Generic Kullanımı

Arayüzlerde jenerik türler kullanarak, esnek ve yeniden kullanılabilir yapı oluşturabilirsiniz.

```
interface Repository<T> {
  getById(id: number): T;
  save(item: T): void;
}
```

36. Class'larda Generic Kullanımı

Sınıflarda jenerik türler kullanarak, tür güvenliğini artırabilirsiniz.

```
class Storage<T> {
  private items: T[] = [];

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

  get(index: number): T {
    return this.items[index];
```

```
  }
}
```