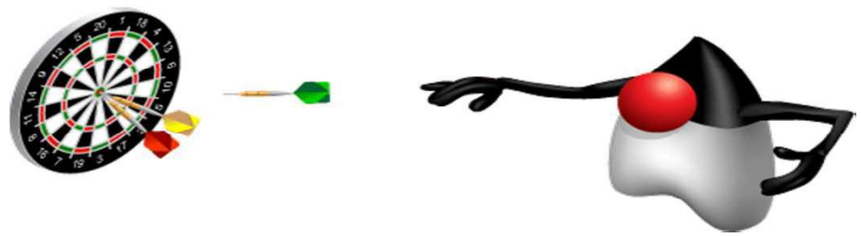
7.1

Typescript Interfaces and Generics

Objectives

After completing this lesson, you should be able to do the following:

- What is TypeScript Interfaces?
- Generics In TypeScript
- TypeScript Decorators



Typescript Interfaces

Interfaces

- Interfaces are used to type-check whether an object fits a certain structure.
- By defining an interface we can name a specific combination of variables, making sure that they will always go together.
- When translated to JavaScript, interfaces disappear - their only purpose is to help in the development stage.

Code Snippet

```
    // Here we define our Food interface, its properties, and their types.
    interface Food
    {
      name: string;
      calories: number;
     }

// We tell our function to expect an object that fulfills the Food interface.
// This way we know that the properties we need will always be available.

    function speak(food: Food): void
    {
      console.log("Our " + food.name + " has " + food.calories      + "
      calories.");
      }
// We define an object that has all of the properties the Food interface
    expects.
// Notice that types will be inferred automatically.

    var ice_cream = { name: "ice cream", calories: 200 }
    speak(ice_cream);
```

```
interface Food {
    name: string;
    calories: number;
}

function speak(food: Food): void{
  console.log("Our " + food.name + " has " + food.calories + " grams.");
}

// We've made a deliberate mistake and name is misspelled as nmae.
var ice_cream = {
  nmae: "ice cream",
  calories: 200
}

speak(ice_cream);
main.ts(16,7): error TS2345: Argument of type '{ nmae: string; calories: number; }
is not assignable to parameter of type 'Food'.
Property 'name' is missing in type '{ nmae: string; calories: number; }'.
```

- Interfaces are used to create contracts.
- They don’t provide a concrete meaning to anything, they just declare the methods and fields.
- Because of this, an interface cannot be used as-is to build anything.
- An interface is meant to be inherited by a class and the class implementing the interface has to define all members of the interface.

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

```
class Square implements IShape{
  constructor(private length: number){}

  get Length(){
    return this.length;
  }

  area(){
    return this.length * this.length;
  }
}


class Rectangle implements IShape{
  constructor(private length: number, private breadth: number){}

  area(): number{
    return this.length * this.breadth;
  }
}
```

- We can instantiate these classes and assign them to references of the interface type.
- We can access the members declared in the interface using this instance.

```
var square: IShape = new Square(10);
var rectangle: IShape = new Rectangle(10, 20);
console.log(square.area());
console.log(rectangle.area());
```

- The class *Square* class has an additional property *Length* that is not declared in the interface.
- Though the object *square* is an instance of *Square* class, we cannot access the members that are defined in the class and not in the interface.
- However, the object *square* can be casted to the type *Square* and then we can use the members defined in the *Square* class.

```
var squareObj = square as Square;
console.log(squareObj.Length);
```

Typescript Generics

- Generics are templates that allow the same function to accept arguments of various different types.
- Creating reusable components using generics is better than using the any data type, as generics preserve the types of the variables that go in and out of them.

```typescript
// The <T> after the function name symbolizes that it's a generic function.
// When we call the function, every instance of T will be replaced with the
// actual provided type.

// Receives one argument of type T,
// Returns an array of type T.

function genericFunc<T>(argument: T): T[] {
  var arrayOfT: T[] = [];    // Create empty array of type T.
  arrayOfT.push(argument);   // Push, now arrayOfT = [argument].
  return arrayOfT;
}

var arrayFromString = genericFunc<string>("beep");
console.log(arrayFromString[0]);          // "beep"
console.log(typeof arrayFromString[0])    // String

var arrayFromNumber = genericFunc(42);
console.log(arrayFromNumber[0]);          // 42
console.log(typeof arrayFromNumber[0])    // number
```

Typescript Decorators

What is AOP ?

- AOP according to wikipedia AOP is a programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns.
- It does so by adding additional behavior to existing code (an advice) without modifying the code itself.
- The biggest advantage of using AOP is the separation of irrelevant code from the main functionality of the method to a different place.

- The best example is logging or authorization. Instead of doing it inside the function which makes it clunky, we pull it out and only declare that the function is doing it.
- In addition, we can change it in a single place rather than in all the code base.

```
// without AOP:
function add(x, y) {
log('foo was called!');
If (!validate(arguments)) {
throw(...)
}
If (!authorized()) {
throw()
}
return x + y;
}

// with AOP
@log
@validate
@authorize
function add(x, y) {
return x + y;
}
```

TypeScript Decorators

- Decorators provide a way to add both annotations and a meta-programming syntax for class declarations and members.
- Decorators are a stage 2 proposal for JavaScript and are available as an experimental feature of TypeScript.

Summary

In this lesson, you should have learned how to:

- What is TypeScript Interfaces?
- Generics In TypeScript
- TypeScript Decorators