TypeScript-2

Function in TypeScript:

Functions in TypeScript are similar to JavaScript functions but with the added benefit of static typing. TypeScript allows you to specify the types of function parameters and return values, making your code more robust and maintainable. Here are five examples of functions in TypeScript:

Example 1 - Basic Function with Type Annotations:

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

const sum: number = add(3, 4);
console.log(sum); // Output: 7
```

In this example, we define a function add that takes two parameters, both of type number, and returns a number. We annotate the parameter and return types for type safety.

Example 2 - Optional Parameters and Default Values:

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

console.log(greet("Alice"));          // Output: "Hello, Alice!"
console.log(greet("Bob", "Hi there")); // Output: "Hi there, Bob!"
```

In this case, the greeting parameter has a default value, making it optional. If no greeting is provided, "Hello" is used.

Example 3 - Rest Parameters:

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

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

The ...numbers syntax represents a rest parameter, allowing you to pass any number of arguments, which are collected into an array.

Example 4 - Function Expressions and Arrow Functions:

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

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

console.log(multiply(3, 4)); // Output: 12
console.log(square(5));      // Output: 25
```
In this example, we define a function expression and an arrow function, both with type annotations.

Example 5 - Function Overloads:

```
function format(value: string): string;
function format(value: number): number;
function format(value: string | number): string | number {
  if (typeof value === "string") {
    return `String: ${value}`;
  } else if (typeof value === "number") {
    return value * 2;
  }
}

console.log(format("Hello")); // Output: "String: Hello"
console.log(format(5));       // Output: 10
```
Here, we define function overloads to handle different types and return values for the format function. The final implementation covers both cases.

Functions in TypeScript allow you to specify and enforce type information for parameters and return values, enhancing code quality and readability. You can use optional parameters, default values, rest parameters, and function overloads to create flexible and type-safe functions.

Function Type Declarations:

Function type declarations in TypeScript allow you to define the shape and type of functions. These declarations specify the types of parameters and return values for functions. Here are five examples of function type declarations in TypeScript:

Example 1 - Basic Function Type Declaration:
```
type AddFunction = (a: number, b: number) => number;
const add: AddFunction = (x, y) => x + y;
const result: number = add(3, 4);
console.log(result); // Output: 7
```
In this example, we declare a function type AddFunction that takes two number parameters and returns a number. The add function adheres to this type.

Example 2 - Function Type with Optional Parameters:
```
type GreetFunction = (name: string, greeting?: string) => string;
const greet: GreetFunction = (name, greeting) => {
  if (greeting) {
    return `${greeting}, ${name}!`;
  }
  return `Hello, ${name}!`;
}

console.log(greet("Alice"));          // Output: "Hello, Alice!"
console.log(greet("Bob", "Hi there")); // Output: "Hi there, Bob!"
```
Here, the function type GreetFunction has an optional parameter for the greeting message.

Example 3 - Function Type with Rest Parameters:
```
type SumFunction = (...numbers: number[]) => number;
const sum: SumFunction = (...numbers) => numbers.reduce((total, num) => total + num, 0);
const total: number = sum(1, 2, 3, 4, 5);
console.log(total); // Output: 15
```
The SumFunction type specifies that the function can accept any number of arguments and returns a number.

Example 4 - Function Type for Callbacks:
```
type CallbackFunction = (data: string) => void;
function processData(data: string, callback: CallbackFunction) {
  callback(data);
}
processData("Hello, TypeScript!", message => {
  console.log(message); // Output: "Hello, TypeScript!"
});
```
In this example, the CallbackFunction type is used to define the type of a callback function passed to processData.

Example 5 - Function Type with Generics:
```
type IdentityFunction<T> = (value: T) => T;
const identity: IdentityFunction<number> = value => value;
const num: number = identity(42);
console.log(num); // Output: 42
```
In this case, the IdentityFunction type uses a generic type parameter to allow the function to work with various data types.

Function type declarations in TypeScript are a powerful tool for defining and enforcing the type of functions, making your code more readable and providing static type checking. They allow you to specify the expected input and output types of functions, which is especially helpful in larger codebases and collaborative development.

Optional and default parameters:

Optional and default parameters in TypeScript allow you to define functions with flexibility by specifying parameters that may not be required or have default values. Here are five examples that demonstrate optional and default parameters in TypeScript:

Example 1 - Optional Parameter:
typescript
Copy code
```
function greet(name: string, greeting?: string): string {
  if (greeting) {
    return `${greeting}, ${name}!`;
  }
  return `Hello, ${name}!`;
}
console.log(greet("Alice"));           // Output: "Hello, Alice!"
console.log(greet("Bob", "Hi there")); // Output: "Hi there, Bob!"
```

In this example, the greeting parameter is optional. If provided, it's used in the greeting message; otherwise, it defaults to "Hello."

Example 2 - Default Parameter Value:

typescript

Copy code

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

console.log(greet("Alice"));          // Output: "Hello, Alice!"
console.log(greet("Bob", "Hi there")); // Output: "Hi there, Bob!"
```

Here, the greeting parameter has a default value of "Hello," so you can omit it when calling the function.

Example 3 - Optional and Default Parameters with Rest Parameter:

```
function printNumbers(...numbers: number[]): void {
  console.log("Numbers:", ...numbers);
}
printNumbers(1, 2, 3);
printNumbers(); // Output: "Numbers:"
```

In this example, the ...numbers rest parameter allows you to pass any number of numbers, and it defaults to an empty array when no arguments are provided.

Example 4 - Combining Optional and Default Parameters:

```
function formatName(firstName: string, lastName?: string, title: string = "Mr."): string {
  if (lastName) {
    return `${title} ${firstName} ${lastName}`;
  }
  return `${title} ${firstName}`;
}
console.log(formatName("John"));          // Output: "Mr. John"
console.log(formatName("Alice", "Smith")); // Output: "Mr. Alice Smith"
console.log(formatName("Bob", undefined)); // Output: "Mr. Bob"
```

This example combines optional and default parameters to handle various name formats.

Example 5 - Using Optional and Default Parameters with Function Type:

```
type GreetFunction = (name: string, greeting?: string) => string;
const greet: GreetFunction = (name, greeting = "Hello") => {
  return `${greeting}, ${name}!`;
}
console.log(greet("Alice"));          // Output: "Hello, Alice!"
console.log(greet("Bob", "Hi there")); // Output: "Hi there, Bob!"
```

This demonstrates that you can use optional and default parameters within function type declarations as well.

Optional and default parameters in TypeScript make your functions more versatile and user-friendly by allowing for flexibility in parameter usage. They are particularly useful when you want to provide a default value for a parameter or make it optional without cluttering your function calls with undefined or null values.

Function overloads :

Function overloads in TypeScript allow you to define multiple function signatures for a single function. Each signature specifies a different set of parameter types and return type. This helps TypeScript provide type-checking based on the specific usage of the function. Here's a detailed explanation with examples:

Example 1 - Basic Function Overload:

```
function greet(name: string): string;
function greet(firstName: string, lastName: string): string;

function greet(a: string, b?: string): string {
  if (b) {
    return `Hello, ${a} ${b}!`;
  }
  return `Hello, ${a}!`;
}

const message1: string = greet("Alice");
const message2: string = greet("Bob", "Smith");
console.log(message1); // Output: "Hello, Alice!"
console.log(message2); // Output: "Hello, Bob Smith!"
```

In this example, the greet function has two overloads that handle different numbers of parameters. The implementation handles both cases based on the provided parameters.

Example 2 - Overloads with Union Types:

```
function format(value: string | number): string;
function format(value: string | number): string {
  if (typeof value === "string") {
    return `String: ${value}`;
  } else {
    return `Number: ${value}`;
  }
}
const result1: string = format("Hello");
const result2: string = format(42);
console.log(result1); // Output: "String: Hello"
console.log(result2); // Output: "Number: 42"
```

In this example, the format function has a single implementation but two overloads to handle both string and number input types.

Example 3 - Function Overloads for Array and Object:

```
function getItem(arr: string[], index: number): string;
function getItem(obj: { [key: string]: string }, key: string): string;

function getItem(data: string[] | { [key: string]: string }, key: number | string): string {
  if (Array.isArray(data)) {
    return data[key as number];
  } else {
    return data[key as string];
  }
}

const arr: string[] = ["apple", "banana", "cherry"];
const obj: { [key: string]: string } = { a: "apple", b: "banana", c: "cherry" };
const item1: string = getItem(arr, 1);
const item2: string = getItem(obj, "b");
console.log(item1); // Output: "banana"
console.log(item2); // Output: "banana"
```
This example demonstrates how to create function overloads for different parameter types and handle them in a single implementation.

Example 4 - Overloads with Generics:
```
function getLength<T>(value: T[]): number;
function getLength(value: string): number;

function getLength(value: any): number {
  if (Array.isArray(value)) {
    return value.length;
  }
  return value.length;
}
const arrLength: number = getLength(["apple", "banana", "cherry"]);
const strLength: number = getLength("typescript");
console.log(arrLength); // Output: 3
console.log(strLength); // Output: 10
```
In this example, the getLength function has two overloads, one for arrays and another for strings, and uses generics for flexibility.

Example 5 - Overloads with Complex Types:
```
type Point = { x: number, y: number };
type Circle = { kind: "circle", center: Point, radius: number };
type Square = { kind: "square", topLeft: Point, sideLength: number };

function area(shape: Circle): number;
function area(shape: Square): number;

function area(shape: Circle | Square): number {
  if (shape.kind === "circle") {
    return Math.PI * shape.radius * shape.radius;
  } else {
    return shape.sideLength ** 2;
  }
}

const circle: Circle = { kind: "circle", center: { x: 0, y: 0 }, radius: 5 };
const square: Square = { kind: "square", topLeft: { x: 0, y: 0 }, sideLength: 4 };

const circleArea: number = area(circle);
const squareArea: number = area(square);

console.log(circleArea); // Output: 78.54
console.log(squareArea); // Output: 16
```
This example shows function overloads for complex types, allowing you to handle different shapes based on the kind property.

Function overloads in TypeScript are a powerful tool for creating type-safe functions with different parameter and return type combinations. They are particularly useful when dealing with functions that can have multiple, distinct behaviors depending on the input.

Classes:

Classes in TypeScript are similar to classes in other object-oriented languages like Java and C#. They allow you to define object blueprints with properties and methods, providing a clear structure for organizing and reusing your code. Here are five examples to demonstrate classes in TypeScript:

Example 1 - Basic Class:

```
typescript
Copy code
class Person {
  firstName: string;
  lastName: string;

  constructor(firstName: string, lastName: string) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

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

const person1 = new Person("John", "Doe");
const person2 = new Person("Alice", "Smith");

console.log(person1.getFullName()); // Output: "John Doe"
console.log(person2.getFullName()); // Output: "Alice Smith"
```

In this example, we define a Person class with properties and methods, then create instances of the class and use its methods.

Example 2 - Inheritance:

```
class Animal {
  name: string;

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

  makeSound(): string {
    return "Some generic sound";
  }
}

class Dog extends Animal {
  makeSound(): string {
    return "Woof!";
    }
}

const dog = new Dog("Buddy");

console.log(dog.makeSound()); // Output: "Woof!"
```
Here, we have a base Animal class, and a derived Dog class that overrides the makeSound method to provide a specific implementation.

Example 3 - Access Modifiers:
```
class Employee {
    private empId: number;
    public fullName: string;

    constructor(empId: number, fullName: string) {
        this.empId = empId;
        this.fullName = fullName;
    }

    getEmployeeInfo(): string {
        return `Employee ID: ${this.empId}, Name: ${this.fullName}`;
    }
}

const employee = new Employee(1, "John Doe");
console.log(employee.fullName); // Accessing public property
console.log(employee.getEmployeeInfo()); // Accessing a method
```
In this example, we use access modifiers like private and public to control access to class members.

Example 4 - Static Members:
```
class MathHelper {
    static PI: number = 3.14;
    static circleArea(radius: number): number {
        return MathHelper.PI * radius * radius;
    }
}

const area = MathHelper.circleArea(5);
console.log(area); // Output: 78.5
```
In this example, we define a class with a static property and a static method, which can be accessed without creating an instance of the class.

Example 5 - Getter and Setter:
```
class Temperature {
    private _celsius: number;

    get celsius(): number {
        return this._celsius;
    }

    set celsius(value: number) {
        if (value < -273.15) {
            throw new Error("Temperature below absolute zero is not possible.");
        }
        this._celsius = value;
    }

    get fahrenheit(): number {
        return (this._celsius * 9/5) + 32;
    }

    constructor(celsius: number) {
        this.celsius = celsius;
    }
}
const temperature = new Temperature(25);
console.log(temperature.celsius); // Accessing the getter
console.log(temperature.fahrenheit); // Accessing the getter for Fahrenheit
temperature.celsius = 30; // Accessing the setter
```

In this example, we use getter and setter methods to control access to temperature properties and provide computed values.

Classes in TypeScript provide a structured way to model your code, allowing you to encapsulate data and behavior. They also support inheritance, access modifiers, static members, and various other features, making them a fundamental part of object-oriented programming in TypeScript.

constructors in TypeScript:

Constructors and properties in TypeScript are essential concepts when working with classes. Constructors are special methods called when you create a new instance of a class, and properties are variables associated with class instances. Let's dive into constructors and properties in TypeScript in detail:

Constructors:

Constructors are used to initialize the state of an object when it's created from a class.
In TypeScript, a constructor is a method named constructor within a class.
You can have only one constructor per class, and it's called when you use the new keyword to create an instance of the class.
Constructors can accept parameters, allowing you to set initial values for class properties.
Example - Constructor with Parameters:

```
class Person {
    firstName: string;
    lastName: string;

    constructor(firstName: string, lastName: string) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
}

const person = new Person("John", "Doe");
console.log(person.firstName); // Output: "John"
console.log(person.lastName); // Output: "Doe"
```

In this example, the Person class has a constructor that accepts firstName and lastName parameters to initialize the object's properties.

Properties:

Properties are variables associated with class instances.
They represent the state of an object and can have access modifiers like public, private, and protected to control their visibility and accessibility.
You can access and modify properties using dot notation (e.g., instance.property).
Example - Public Property:

```
class Rectangle {
  width: number;
  height: number;

  constructor(width: number, height: number) {
    this.width = width;
    this.height = height;
  }

  getArea(): number {
    return this.width * this.height;
  }
}

const rect = new Rectangle(5, 10);
console.log(rect.width); // Output: 5
console.log(rect.height); // Output: 10
console.log(rect.getArea()); // Output: 50
```

In this example, the Rectangle class has two public properties, width and height, which are accessed and used within the class's methods.

Example - Private Property:

```
class BankAccount {
  private balance: number;

  constructor(initialBalance: number) {
    this.balance = initialBalance;
  }

  deposit(amount: number): void {
    this.balance += amount;
  }

  withdraw(amount: number): void {
    if (amount <= this.balance) {
      this.balance -= amount;
    } else {
      console.log("Insufficient balance.");
    }
  }
}
```
In this example, the balance property is marked as private, meaning it can only be accessed within the BankAccount class. This encapsulates the balance, preventing external modification.

Accessors (Getter and Setter):

TypeScript allows you to define getter and setter methods to control property access and modification.
Getters are used to retrieve the property value, and setters are used to change it.
They provide a way to perform additional logic when accessing or modifying properties.
Example - Getter and Setter:
```
class Temperature {
   private _celsius: number;

   get celsius(): number {
      return this._celsius;
   }

   set celsius(value: number) {
      if (value < -273.15) {
         throw new Error("Temperature below absolute zero is not possible.");
      }
      this._celsius = value;
   }

   constructor(celsius: number) {
      this.celsius = celsius;
   }
}

const temperature = new Temperature(25);
console.log(temperature.celsius); // Accessing the getter
temperature.celsius = 30; // Accessing the setter
```
In this example, the Temperature class uses a getter and setter for the celsius property to ensure the temperature remains within a valid range.

Constructors and properties in TypeScript play a crucial role in defining and initializing class instances. They allow you to encapsulate data and logic within classes, making your code more organized and maintainable.

In TypeScript, inheritance allows you to create a new class that inherits properties and methods from an existing class, referred to as the base or parent class. Abstract classes are a special type of class that cannot be instantiated directly but can be used as a blueprint for other classes. Let's delve into inheritance and abstract classes with examples:

Inheritance:

Inheritance in TypeScript involves creating a derived or child class that inherits members (properties and methods) from a base or parent class. The derived class can also have additional members or override inherited members. The primary keyword for inheritance is extends.

Example 1 - Basic Inheritance:
```
class Animal {
   name: string;
```

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

  makeSound(): void {
    console.log("Some generic sound");
  }
}

class Dog extends Animal {
  makeSound(): void {
    console.log("Woof!");
  }
}

const dog = new Dog("Buddy");
console.log(dog.name);     // Output: "Buddy"
dog.makeSound();           // Output: "Woof!"
```

In this example, the Dog class inherits from the Animal class and overrides the makeSound method.

Example 2 - Accessing Superclass Members:

```
class Vehicle {
  protected speed: number;

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

  accelerate(acceleration: number): void {
    this.speed += acceleration;
  }
}

class Car extends Vehicle {
  constructor(speed: number) {
    super(speed); // Call the base class constructor
  }

  accelerate(acceleration: number): void {
    super.accelerate(acceleration);
    console.log(`Car is now moving at ${this.speed} km/h`);
  }
}

const myCar = new Car(60);
myCar.accelerate(20); // Output: "Car is now moving at 80 km/h"
```

In this example, the Car class uses the super keyword to call the constructor and methods of the base class.

Abstract Classes:

Abstract classes in TypeScript are designed to be used as base classes for other classes but cannot be instantiated directly. They can contain abstract methods that must be implemented by derived classes. The abstract keyword is used to define an abstract class or method.

Example 3 - Abstract Class and Method:

typescript
Copy code
```
abstract class Shape {
  abstract calculateArea(): number;
}

class Circle extends Shape {
  radius: number;

  constructor(radius: number) {
    super();
    this.radius = radius;
  }

  calculateArea(): number {
    return Math.PI * this.radius ** 2;
  }
}

const circle = new Circle(5);
console.log(circle.calculateArea()); // Output: 78.54
```
In this example, the Shape class is abstract with an abstract method calculateArea(). The Circle class extends Shape and provides an implementation for calculateArea.

Example 4 - Abstract Properties:

```
abstract class Employee {
  abstract name: string;
  abstract role: string;

  abstract getSalary(): number;
}

class Manager extends Employee {
  name: string;
  role: string;

  constructor(name: string, role: string) {
    super();
    this.name = name;
    this.role = role;
  }

  getSalary(): number {
    // Implement the logic to calculate manager's salary
    return 60000;
    }
}

const manager = new Manager("Alice", "Project Manager");
console.log(manager.name);   // Output: "Alice"
console.log(manager.role);   // Output: "Project Manager"
console.log(manager.getSalary()); // Output: 60000
```

In this example, the Employee class defines abstract properties and an abstract method. The Manager class extends Employee and provides implementations for these members.

Summary:

Inheritance and abstract classes are fundamental concepts in object-oriented programming with TypeScript. Inheritance allows you to create class hierarchies and share common properties and methods, while abstract classes provide a blueprint for derived classes and ensure that certain methods or properties are implemented. These features help you structure your code in a more organized and maintainable way.

Acess Modifiers:

Access modifiers in TypeScript provide control over the visibility and accessibility of class members (properties and methods) within the class and from external code. TypeScript supports three main access modifiers:

public: The default access modifier. Members marked as public are accessible from anywhere, both within the class and externally.

private: Members marked as private are only accessible within the defining class. They cannot be accessed or modified from outside the class.

protected: Members marked as protected are accessible within the defining class and its subclasses (derived classes). They cannot be accessed from external code.

Here are detailed examples for each access modifier:

Example 1 - Public Access Modifier:

```
class Car {
    public brand: string;

    constructor(brand: string) {
        this.brand = brand;
    }

    start(): void {
        console.log(`Starting the ${this.brand} car.`);
    }
}

const myCar = new Car("Toyota");
console.log(myCar.brand); // Accessing public property
myCar.start(); // Accessing public method
```

In this example, brand and start are marked as public, making them accessible from external code.

Example 2 - Private Access Modifier:

```
class BankAccount {
  private balance: number;

  constructor(initialBalance: number) {
    this.balance = initialBalance;
  }

  deposit(amount: number): void {
    this.balance += amount;
  }

  withdraw(amount: number): void {
    if (amount <= this.balance) {
      this.balance -= amount;
    } else {
      console.log("Insufficient balance.");
    }
  }
}

const account = new BankAccount(1000);
account.deposit(500); // Accessing private method
console.log(account.balance); // Error: Property 'balance' is private
```

In this example, balance is marked as private, so it can only be accessed within the BankAccount class.

Example 3 - Protected Access Modifier:

```
class Person {
  protected age: number;

  constructor(age: number) {
    this.age = age;
  }
}

class Student extends Person {
  constructor(age: number) {
    super(age);
  }

  getAge(): number {
    return this.age; // Accessing protected property in a subclass
  }
}

const student = new Student(20);
console.log(student.getAge()); // Accessing protected method in a subclass
```

In this example, age is marked as protected in the Person class, making it accessible in the derived Student class.

Access modifiers are important for encapsulation and maintaining the integrity of your classes. They help you control how class members are accessed and modified, which is crucial for building robust and maintainable software.