### TypeScript Coding Test Questions

1. \*\*Create a Class with Methods\*\*

\*\*Question:\*\* Create a class `BankAccount` with properties `accountNumber` (number) and `balance` (number). Add methods `deposit` and `withdraw` to update the balance.

```typescript

class BankAccount {

accountNumber: number;

balance: number;

constructor(accountNumber: number, initialBalance: number) {

this.accountNumber = accountNumber;

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 funds');

}

}

}

// Usage

const account = new BankAccount(12345, 1000);

account.deposit(500);

account.withdraw(200);

console.log(account.balance); // Output: 1300

```

2. \*\*Inheritance and Method Overriding\*\*

\*\*Question:\*\* Create a base class `Animal` with a method `makeSound`. Create a subclass `Dog` that overrides the `makeSound` method to print "Woof!".

```typescript

class Animal {

makeSound(): void {

console.log("Some sound");

}

}

class Dog extends Animal {

makeSound(): void {

console.log("Woof!");

}

}

// Usage

const dog = new Dog();

dog.makeSound(); // Output: Woof!

```

3. \*\*Constructor Overloading (Optional Parameters)\*\*

\*\*Question:\*\* Create a class `Rectangle` with properties `width` and `height`. Allow the constructor to accept either one or two parameters. If one parameter is provided, set both `width` and `height` to that value (i.e., create a square).

```typescript

class Rectangle {

width: number;

height: number;

constructor(width: number, height?: number) {

this.width = width;

this.height = height !== undefined ? height : width;

}

area(): number {

return this.width \* this.height;

}

}

// Usage

const square = new Rectangle(5);

const rectangle = new Rectangle(5, 10);

console.log(square.area()); // Output: 25

console.log(rectangle.area()); // Output: 50

```

4. \*\*Static Methods and Properties\*\*

\*\*Question:\*\* Create a class `MathUtils` with a static method `square` that returns the square of a number and a static property `PI` with the value 3.14.

```typescript

class MathUtils {

static PI: number = 3.14;

static square(num: number): number {

return num \* num;

}

}

// Usage

console.log(MathUtils.PI); // Output: 3.14

console.log(MathUtils.square(4)); // Output: 16

```

5. \*\*Interface Implementation\*\*

\*\*Question:\*\* Create an interface `Shape` with a method `area`. Implement this interface in classes `Circle` and `Square`.

```typescript

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;

}

}

class Square implements Shape {

side: number;

constructor(side: number) {

this.side = side;

}

area(): number {

return this.side \* this.side;

}

}

// Usage

const circle = new Circle(5);

const square = new Square(4);

console.log(circle.area()); // Output: 78.53981633974483

console.log(square.area()); // Output: 16

```

6. \*\*Private Properties and Methods\*\*

\*\*Question:\*\* Create a class `Person` with private properties `firstName` and `lastName`, and a private method `getFullName`. Add a public method `introduce` that uses `getFullName`.

```typescript

class Person {

private firstName: string;

private lastName: string;

constructor(firstName: string, lastName: string) {

this.firstName = firstName;

this.lastName = lastName;

}

private getFullName(): string {

return `${this.firstName} ${this.lastName}`;

}

public introduce(): void {

console.log(`Hello, my name is ${this.getFullName()}`);

}

}

// Usage

const person = new Person('John', 'Doe');

person.introduce(); // Output: Hello, my name is John Doe

```

7. \*\*Readonly Properties\*\*

\*\*Question:\*\* Create a class `Book` with readonly properties `title` and `author`.

```typescript

class Book {

readonly title: string;

readonly author: string;

constructor(title: string, author: string) {

this.title = title;

this.author = author;

}

}

// Usage

const book = new Book('1984', 'George Orwell');

console.log(book.title); // Output: 1984

console.log(book.author); // Output: George Orwell

```

8. \*\*Getters and Setters\*\*

\*\*Question:\*\* Create a class `Temperature` with a private property `\_celsius`. Add getter and setter for the `fahrenheit` property.

```typescript

class Temperature {

private \_celsius: number;

constructor(celsius: number) {

this.\_celsius = celsius;

}

get fahrenheit(): number {

return this.\_celsius \* 9/5 + 32;

}

set fahrenheit(value: number) {

this.\_celsius = (value - 32) \* 5/9;

}

}

// Usage

const temp = new Temperature(0);

console.log(temp.fahrenheit); // Output: 32

temp.fahrenheit = 100;

console.log(temp.fahrenheit); // Output: 100

```

9. \*\*Abstract Classes and Methods\*\*

\*\*Question:\*\* Create an abstract class `Vehicle` with an abstract method `move`. Create a subclass `Car` that implements the `move` method.

```typescript

abstract class Vehicle {

abstract move(): void;

}

class Car extends Vehicle {

move(): void {

console.log('The car is moving');

}

}

// Usage

const car = new Car();

car.move(); // Output: The car is moving

```

10. \*\*Class with Optional Properties\*\*

\*\*Question:\*\* Create a class `Employee` with properties `name` and optional `department`.

```typescript

class Employee {

name: string;

department?: string;

constructor(name: string, department?: string) {

this.name = name;

if (department) {

this.department = department;

}

}

}

// Usage

const emp1 = new Employee('Alice');

const emp2 = new Employee('Bob', 'Engineering');

console.log(emp1); // Output: Employee { name: 'Alice' }

console.log(emp2); // Output: Employee { name: 'Bob', department: 'Engineering' }

```

11. \*\*Class with Default Parameters\*\*

\*\*Question:\*\* Create a class `Product` with properties `name` and `price`. The constructor should have a default value for `price`.

```typescript

class Product {

name: string;

price: number;

constructor(name: string, price: number = 100) {

this.name = name;

this.price = price;

}

}

// Usage

const prod1 = new Product('Gadget');

const prod2 = new Product('Widget', 200);

console.log(prod1); // Output: Product { name: 'Gadget', price: 100 }

console.log(prod2); // Output: Product { name: 'Widget', price: 200 }

```

12. \*\*Class with Method Overloading\*\*

\*\*Question:\*\* Create a class `Logger` with an overloaded method `log` that can take either a string or a number.

```typescript

class Logger {

log(message: string): void;

log(value: number): void;

log(value: any): void {

console.log(value);

}

}

// Usage

const logger = new Logger();

logger.log('Hello'); // Output: Hello

logger.log(123); // Output: 123

```

13. \*\*Class with Interface Implementation\*\*

\*\*Question:\*\* Create an interface `Flyable` with a method `fly`. Create a class `Bird` that implements this interface.

```typescript

interface Flyable {

fly(): void;

}

class Bird implements Flyable {

fly(): void {

console.log('The bird is flying');

}

}

// Usage

const bird = new Bird();

bird.fly(); // Output: The bird is flying

```

14. \*\*Class with Static Properties and Methods\*\*

\*\*Question:\*\* Create a class `Counter` with a static property `count` and a static method `increment`.

```typescript

class Counter {

static count: number = 0;

static increment(): void {

this.count++;

}

}

// Usage

Counter.increment();

Counter.increment();

console.log(Counter.count); // Output: 2

```

15. \*\*Polymorphism with Classes\*\*

\*\*Question:\*\* Create a base class `Shape` with a method `area`. Create two subclasses `Rectangle` and `Circle` that override the `area` method.

```typescript

class Shape {

area(): number {

return 0;

}

}

class Rectangle extends Shape {

width: number;

height: number;

constructor(width: number, height: number) {

super();

this.width = width;

this.height = height;

}

area(): number {

return this.width \* this.height;

}

}

class Circle extends Shape {

radius: number;

constructor(radius: number) {

super();

this.radius = radius;

}

area(): number {

return Math.PI \* this.radius \* this.radius;

}

}

// Usage

const rect = new Rectangle(10, 20);

const circle = new Circle(5);

console.log(rect.area()); // Output: 200

console.log(circle.area()); // Output: 78.53981633974483

```

16. \*\*Interface with Optional Properties\*\*

\*\*Question:\*\* Create an interface `Vehicle` with properties `make` and optional `year`.

```typescript

interface Vehicle {

make: string;

year?: number;

}

const car: Vehicle = {

make: 'Toyota',

year: 2020

};

const bike: Vehicle = {

make: 'Honda'

};

console.log(car); // Output: { make: 'Toyota', year: 2020 }

console.log(bike); // Output: { make: 'Honda' }

```

17. \*\*Abstract Class with Abstract Method\*\*

\*\*Question:\*\* Create an abstract class `Employee` with an abstract method `calculateSalary`. Create two subclasses `FullTimeEmployee` and `PartTimeEmployee` that implement the `calculateSalary` method.

```typescript

abstract class Employee {

abstract calculateSalary(): number;

}

class FullTimeEmployee extends Employee {

salary: number;

constructor(salary: number) {

super();

this.salary = salary;

}

calculateSalary(): number {

return this.salary;

}

}

class PartTimeEmployee extends Employee {

hourlyRate: number;

hoursWorked: number;

constructor(hourlyRate: number, hoursWorked: number) {

super();

this.hourlyRate = hourlyRate;

this.hoursWorked = hoursWorked;

}

calculateSalary(): number {

return this.hourlyRate \* this.hoursWorked;

}

}

// Usage

const fullTime = new FullTimeEmployee(5000);

const partTime = new PartTimeEmployee(20, 80);

console.log(fullTime.calculateSalary()); // Output: 5000

console.log(partTime.calculateSalary()); // Output: 1600

```

18. \*\*Readonly Property with Interface\*\*

\*\*Question:\*\* Create an interface `Person` with a readonly property `id` and a property `name`.

```typescript

interface Person {

readonly id: number;

name: string;

}

const person: Person = {

id: 1,

name: 'John Doe'

};

console.log(person); // Output: { id: 1, name: 'John Doe' }

// person.id = 2; // Error: Cannot assign to 'id' because it is a read-only property.

```

19. \*\*Method Chaining\*\*

\*\*Question:\*\* Create a class `Calculator` with methods `add`, `subtract`, `multiply`, and `divide` that allow method chaining.

```typescript

class Calculator {

private value: number;

constructor(initialValue: number = 0) {

this.value = initialValue;

}

add(amount: number): this {

this.value += amount;

return this;

}

subtract(amount: number): this {

this.value -= amount;

return this;

}

multiply(amount: number): this {

this.value \*= amount;

return this;

}

divide(amount: number): this {

if (amount !== 0) {

this.value /= amount;

}

return this;

}

getResult(): number {

return this.value;

}

}

// Usage

const result = new Calculator(10)

.add(5)

.subtract(3)

.multiply(4)

.divide(2)

.getResult();

console.log(result); // Output: 24

```

20. \*\*Singleton Pattern\*\*

\*\*Question:\*\* Implement a singleton pattern for a class `Database`.

```typescript

class Database {

private static instance: Database;

private constructor() {

// Private constructor to prevent direct instantiation

}

static getInstance(): Database {

if (!Database.instance) {

Database.instance = new Database();

}

return Database.instance;

}

query(sql: string): void {

console.log(`Executing query: ${sql}`);

}

}

// Usage

const db1 = Database.getInstance();

const db2 = Database.getInstance();

db1.query('SELECT \* FROM users'); // Output: Executing query: SELECT \* FROM users

console.log(db1 === db2); // Output: true

```

These questions focus on object-oriented programming concepts in TypeScript, such as classes, interfaces, inheritance, method overloading, and design patterns. They are designed to test a beginner's understanding of TypeScript's core features and object-oriented principles.