30 Ready■Made TypeScript Programs

Beginner + Intermediate + Advanced (Professional Theme)

1. Beginner 1 – Add Two Numbers

Shows a basic typed function in TypeScript with numeric parameters and return type.

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

console.log("Sum =", add(10, 20));
```

2. Beginner 2 – Check Prime Number

Checks if a number is prime using a loop till square root of the number for better efficiency.

```
function isPrime(num: number): boolean {
    if (num <= 1) return false;
    for (let i = 2; i <= Math.sqrt(num); i++) {
        if (num % i === 0) return false;
    }
    return true;
}

console.log(isPrime(11) ? "Prime" : "Not Prime");
```

3. Beginner 3 – Reverse a String

Uses split, reverse and join to reverse a string.

```
function reverseString(str: string): string {
    return str.split("").reverse().join("");
}

console.log(reverseString("Thanesh"));
```

4. Beginner 4 – Count Vowels in String

Demonstrates filtering characters based on a set of vowels.

```
function countVowels(str: string): number {
    const vowels = "aeiouAEIOU";
    return str.split("").filter(ch => vowels.includes(ch)).length;
```

```
}

console.log(countVowels("Hello World"));
```

5. Beginner 5 – Print Even Numbers from Array

Shows use of typed array and the filter() method with an arrow function.

```
let nums: number[] = [10, 21, 32, 43, 54];

let even = nums.filter(n => n % 2 === 0);
console.log(even);
```

6. Beginner 6 – Find Largest of Three Numbers

Uses Math.max to get the largest among three numbers.

```
function largest(a: number, b: number, c: number): number {
    return Math.max(a, b, c);
}

console.log(largest(12, 99, 45));
```

7. Beginner 7 – String Palindrome Check

Checks if a string reads the same forwards and backwards.

```
function isPalindrome(str: string): boolean {
    return str === str.split("").reverse().join("");
}

console.log(isPalindrome("madam"));
```

8. Beginner 8 – Simple Employee Interface

Introduces TypeScript interfaces for object shape enforcement.

```
interface Employee {
    id: number;
    name: string;
    salary: number;
}

const emp: Employee = {
    id: 101,
    name: "Kajal",
    salary: 50000
};
```

```
console.log(emp);
```

9. Beginner 9 – Student Class with Marks

Shows a simple class with constructor and instance method.

```
class Student {
    name: string;
    marks: number;

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

    display() {
        console.log(`${this.name} scored ${this.marks}`);
    }
}

let obj = new Student("Megha", 95);
obj.display();
```

10. Beginner 10 – Sum of Digits of a Number

Uses while loop and modulo to compute sum of digits.

```
function sumOfDigits(n: number): number {
    let sum = 0;
    while (n > 0) {
        sum += n % 10;
        n = Math.floor(n / 10);
    }
    return sum;
}

console.log(sumOfDigits(987));
```

11. Intermediate 1 – Employee Class with Bonus Calculation

Illustrates class with typed constructor, methods, and basic business logic.

```
class Employee {
    constructor(
        public id: number,
        public name: string,
        public salary: number
    ) {}

    getAnnualSalary(): number {
        return this.salary * 12;
    }

    giveBonus(percent: number): number {
        return this.salary + (this.salary * percent / 100);
    }
}

const e = new Employee(101, "Lavanya", 50000);
console.log("Annual:", e.getAnnualSalary());
console.log("After Bonus:", e.giveBonus(10));
```

12. Intermediate 2 – Product Interface with Filter

Real-time style example filtering costly products using interfaces and arrays.

```
interface Product {
    id: number;
    name: string;
    price: number;
}

let products: Product[] = [
    {id: 1, name: "Laptop", price: 55000},
    {id: 2, name: "Mouse", price: 500},
    {id: 3, name: "Keyboard", price: 1500}
];

let costly = products.filter(p => p.price > 2000);
console.log(costly);
```

13. Intermediate 3 – Class Inheritance (Person & Developer)

Shows inheritance, constructor chaining and a method using parent fields.

```
class Person {
    constructor(public name: string) {}
}

class Developer extends Person {
    constructor(name: string, public language: string) {
        super(name);
    }

    info() {
        console.log(`${this.name} codes in ${this.language}`);
    }
}

new Developer("Thanesh", "TypeScript").info();
```

14. Intermediate 4 – Function Overloading

Two different parameter types handled via a single implementation using union type.

```
function show(x: number): void;
function show(x: string): void;

function show(x: any) {
    console.log("Value:", x);
}

show(10);
show("Hello TS");
```

15. Intermediate 5 – Remove Duplicates from Array

Uses Set and spread operator to create an array of unique values.

```
let items = [1, 2, 2, 3, 3, 4, 5];
let unique = [...new Set(items)];

console.log(unique);
```

16. Intermediate 6 – Promise Example (API Call Simulation)

Simulates asynchronous data fetch using Promise and setTimeout.

```
function fetchData(): Promise {
    return new Promise((resolve) => {
        setTimeout(() => resolve("Data received"), 1500);
    });
}
```

```
fetchData().then(res => console.log(res));
```

17. Intermediate 7 – Async / Await Example

Demonstrates async function and awaiting a Promise result.

```
async function getUser() {
    return "User Loaded";
}

async function start() {
    let result = await getUser();
    console.log(result);
}

start();
```

18. Intermediate 8 – Map for Key/Value Storage

Uses ES6 Map with typed keys and values.

```
let map = new Map();
map.set(1, "India");
map.set(2, "USA");

console.log(map.get(2));
```

19. Intermediate 9 – Generic Function identity

Shows generics to maintain the type of argument and return value.

```
function identity(value: T): T {
    return value;
}

console.log(identity(100));
console.log(identity("TS"));
```

20. Intermediate 10 – Sort Students by Marks

Sorts an array of objects using a custom comparator.

```
interface Student {
    name: string;
    marks: number;
}

let list: Student[] = [
```

```
    {name: "A", marks: 90},
    {name: "B", marks: 75},
    {name: "C", marks: 85}
];

let sorted = list.sort((a, b) => b.marks - a.marks);
console.log(sorted);
```

21. Advanced 1 – Abstract Class & Polymorphism

Uses an abstract base class with multiple concrete implementations and polymorphic calls.

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

class Circle extends Shape {
    constructor(private r: number) {
        super();
    }
    area(): number {
        return 3.14 * this.r * this.r;
    }
}

class Rectangle extends Shape {
    constructor(private w: number, private h: number) {
        super();
    }
    area(): number {
        return this.w * this.h;
    }
}

let shapes: Shape[] = [new Circle(5), new Rectangle(4, 6)];

shapes.forEach(s => console.log("Area:", s.area()));
```

22. Advanced 2 – Generic Pair Class

Demonstrates generics with a class that holds two different typed values.

```
class Pair {
    constructor(public x: X, public y: Y) {}
}

let obj = new Pair(10, "Ten");
console.log(obj);
```

23. Advanced 3 – Class Decorator Example

Shows a simple class decorator that logs when a class is defined.

```
function Log(constructor: Function) {
    console.log("Class Decorator Applied!");
}

@Log
class Demo {}
```

24. Advanced 4 – File Upload Simulation with Async/Await

Simulates uploading a file and handling completion using async/await.

```
async function uploadFile(file: string): Promise {
    return new Promise(resolve => {
        setTimeout(() => resolve(file + " uploaded"), 2000);
    });
}

async function startUpload() {
    let result = await uploadFile("resume.pdf");
    console.log(result);
}

startUpload();
```

25. Advanced 5 – Dependency Injection Style Example

Shows passing a Logger dependency into a service class, similar to DI in frameworks.

```
class Logger {
    log(msg: string) {
        console.log("LOG:", msg);
    }
}

class UserService {
    constructor(private logger: Logger) {}

    saveUser() {
        this.logger.log("User saved");
    }
}

let service = new UserService(new Logger());
service.saveUser();
```

26. Advanced 6 – Tuple Example

Uses a fixed-length tuple with different types.

```
let emp: [number, string, number] = [101, "Megha", 90000];
console.log(emp);
```

27. Advanced 7 – Intersection Types

Combines two types into one using intersection (&).

```
type A = { name: string };
type B = { age: number };

type Person = A & B;

let p: Person = { name: "Thanesh", age: 30 };
console.log(p);
```

28. Advanced 8 – Order System (Mini In■Memory Project)

Implements a small order service with total bill calculation logic.

```
interface Order {
    id: number;
    item: string;
    qty: number;
    price: number;
}

class OrderService {
    private orders: Order[] = [];

    create(order: Order) {
        this.orders.push(order);
    }

    totalAmount(): number {
        return this.orders
                   .map(o => o.qty * o.price)
                   .reduce((a, b) => a + b, 0);
    }
}

let os = new OrderService();

os.create({id: 1, item: "Pizza", qty: 2, price: 250});
os.create({id: 2, item: "Burger", qty: 3, price: 150});

console.log("Total Bill =", os.totalAmount());
```

29. Advanced 9 – Advanced Async API Simulation

Simulates success/failure using Promise resolve / reject and handles it with then/catch.

```
function apiCall(): Promise {
    return new Promise((resolve, reject) => {
```

```
        let success = true;
        success ? resolve("Success!") : reject("Failed!");
    });
}

apiCall()
    .then(res => console.log(res))
    .catch(err => console.error(err));
```

30. Advanced 10 – Complete Student Management (CRUD In■Memory)

A small CRUD-style service managing students in an in-memory array.

```
interface Student {
    id: number;
    name: string;
    marks: number;
}

class StudentService {
    private students: Student[] = [];

    add(student: Student) {
        this.students.push(student);
    }

    getAll() {
        return this.students;
    }

    getById(id: number) {
        return this.students.find(s => s.id === id);
    }

    update(id: number, marks: number) {
        let s = this.students.find(st => st.id === id);
        if (s) s.marks = marks;
    }

    delete(id: number) {
        this.students = this.students.filter(s => s.id !== id);
    }
}

let ss = new StudentService();
ss.add({id: 1, name: "A", marks: 90});
ss.add({id: 2, name: "B", marks: 80});
ss.update(2, 95);

console.log(ss.getAll());
```