1)What is TypeScript and why is it used instead of JavaScript?
Answer:
TypeScript is a superset of JavaScript that adds static typing.
It is used instead of JavaScript to catch errors at compile time, improve code readability, provide better IDE support, and make large automation frameworks more maintainable and scalable.
let age: number = 30;
// age = "30"; ✕ Compile-time error

2) What are the key benefits of using TypeScript in test automation?
Answer:
TypeScript helps detect errors early before execution.
It improves maintainability of Page Object Models.
Provides strong typing, better refactoring, and reduces runtime failures in automation tests.
class LoginPage {
login(username: string, password: string): void {
console.log(username, password);
}
}

3) Difference between any, unknown, and never?
Answer:
any disables type checking completely and should be avoided.
unknown is safer than any and requires type validation before usage.
never represents values that never occur, such as functions that throw errors or never return.
let a: any = 10;
let b: unknown = "text";
function error(): never {
throw new Error("Error");
}

4) What are primitive types and non-primitive data in TypeScript?
Answer:
Primitive types represent basic values.
They include string, number, boolean, null, undefined, symbol, and bigint.
let name: string = "Shrinivas";
let active: boolean = true;
Non-primitive types store reference values instead of single values.
They include arrays, objects, tuples, enums, and functions.
// Array
let numbers: number[] = [1, 2, 3];
// Object
let user: { name: string; age: number } = {
name: "Shrinivas",
age: 30
};
// Tuple
let userTuple: [number, string] = [1, "Shrinivas"];
// Enum
enum Role {
Admin,
User,
Guest
}
let role: Role = Role.Admin;
// Function
function add(a: number, b: number): number {
return a + b;
}

5) Difference between null and undefined?
Answer:
null represents an intentional absence of value and must be explicitly assigned.
undefined means a variable is declared but not assigned any value.
undefined is the default value for uninitialized variables.
let a = null;
let b: string | undefined;

6) What is type inference?
Answer:
Type inference is TypeScript’s ability to automatically determine the data type of a variable based on its assigned value, without explicitly specifying the type.
let value: any = "Hello";
if (typeof value === "string") {

```
console.log("Value is a string");
}
typeof "abc" // "string"
typeof 10 // "number"
typeof true // "boolean"
typeof {} // "object"
typeof [] // "object"
typeof null // "object" !
typeof undefined // "undefined"
typeof function(){} // "function"
```

7) What is explicit vs implicit typing?
Answer:
Explicit typing is when the type is manually defined.
Implicit typing is when TypeScript infers the type.

```
let x: number = 10; // explicit
let y = 20; // implicit
```

8) What does strict mode do in TypeScript?
Answer:
Strict mode enables all strict type-checking options.
It prevents common runtime errors, enforces better coding practices, and ensures maximum type safety in applications.

```
{
"compilerOptions": {
"strict": true
}
}
```

9) Difference between number and Number?
Answer:
number is a primitive data type and is recommended.
Number is an object wrapper and should be avoided.
Using Number can cause unexpected behavior and performance issues.

```
let a: number = 10;
let b: Number = new Number(10);
```

10) What is void and when do you use it?
Answer:
void indicates that a function does not return any value.
It is commonly used for logging functions, setup methods, and test hooks where no return value is required.

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

11) Difference between let, const, and var in TypeScript?
Answer:
var is function-scoped and can be redeclared, which may cause bugs.
let is block-scoped and allows reassignment but not redeclaration in the same scope.
const is block-scoped and does not allow reassignment, making code safer and more predictable.

```
let count = 1;
const max = 10;
// max = 20; ✕
```

12) How do you define a variable with multiple types?
Answer:
A variable with multiple types is defined using a union type with the | operator.
It allows the variable to hold values of different specified types.

```
let id: number | string;
id = 1;
id = "shree";
```

13) What is a union type?
Answer:
A union type allows a variable to hold one of multiple types.
Code Example:

```
let status: "pass" | "fail";
status = "pass";
```

14) What is an intersection type?
Answer:
An intersection type combines multiple types into one.
Code Example:

```
type User = { name: string };
```

type Admin = { role: string };
type AdminUser = User & Admin;

15) What is a literal type?

Answer:
Literal types restrict variables to exact values.
Code Example:
let role: "admin" | "user";
role = "admin";

16) What is a tuple?

Answer:
A tuple is a fixed-length array with predefined types.
Code Example:
let user: [string, number] = ["Ram", 30];

17) Difference between tuple and array?

Answer:
Array has same type elements.
Tuple has fixed length and different types.
Code Example:
let arr: number[] = [1, 2];
let tup: [string, number] = ["A", 1];

18) How do you define readonly properties?

Answer: readonly properties are properties whose value cannot be changed after initialization.
Using the readonly keyword.
Code Example:
interface User {
readonly id: number;
}

19) What is as const?

Answer:
as const makes object properties readonly and literal.
Code Example:
const config = {
env: "prod",
timeout: 5000
} as const;

20) How do you handle optional properties?

Answer:
Using the ? symbol.
Code Example:
interface Profile {
name: string;
age?: number;
}

21) How do you define function return types?

Answer:
You define a function’s return type by specifying it after the parameters using :.
It ensures the function returns the expected type.
Code Example:
function add(a: number, b: number): number {
return a + b;
}
const result: number = add(5, 10);

22) Difference between void and never in functions?

Answer:
void is used when a function does not return any value.
never is used when a function never returns (e.g., throws an error or has an infinite loop).
Code Example:
function logMessage(msg: string): void {
console.log(msg);
}
function throwError(): never {
throw new Error("This function never returns");
}

23) What are optional parameters?

Answer:

Optional parameters are parameters that may or may not be provided.
They are defined using ? after the parameter name.
Code Example:

```
function greet(name: string, title?: string) {
if (title) {
console.log(`Hello ${title} ${name}`);
} else {
console.log(`Hello ${name}`);
}
}
greet("Shrinivas"); // Hello Shrinivas
greet("Shrinivas", "Mr."); // Hello Mr. Shrinivas
```

24) What are default parameters?
Answer:
Default parameters provide a value if the caller does not pass an argument.
Code Example:

```
function greet(name: string, title: string = "Mr.") {
console.log(`Hello ${title} ${name}`);
}
greet("Shrinivas"); // Hello Mr. Shrinivas
```

25) How do you type arrow functions?
Answer:
Arrow functions can have typed parameters and return type using =>.
Code Example:

```
const multiply = (a: number, b: number): number => a * b;
console.log(multiply(5, 3)); // 15
```

26) What is function overloading?
Answer:
Function overloading allows multiple function signatures for the same function name.
TypeScript decides which implementation to use based on parameter types.
Code Example:

```
function add(a: number, b: number): number;
function add(a: string, b: string): string;
function add(a: any, b: any): any {
return a + b;
}
console.log(add(1, 2)); // 3
console.log(add("A", "B")); // AB
```

27) How do you overload a function in TypeScript?
Answer:
Define multiple function signatures, followed by a single implementation that handles all cases.
Code Example:

```
function display(x: number): void;
function display(x: string): void;
function display(x: any): void {
console.log("Value:", x);
}
display(10);
display("Hello");
```

28) What is a rest parameter?
Answer:
A rest parameter collects all remaining arguments into an array.
It allows functions to accept variable number of arguments.
Code Example:

```
function sum(...numbers: number[]): number {
return numbers.reduce((acc, curr) => acc + curr, 0);
}
console.log(sum(1, 2, 3, 4)); // 10
```

29) What is a callback type?
Answer:
A callback type specifies the type of a function passed as a parameter.
Code Example:

```
function processData(callback: (data: string) => void) {
const data = "Hello World";
callback(data);
}
processData((msg) => console.log(msg)); // Hello World
```

30) How do you type async functions?
Answer:
Async functions always return a Promise. You type the resolved value using Promise<type>.
Code Example:
```
async function fetchData(): Promise<string> {
return "Data fetched";
}
fetchData().then((res) => console.log(res)); // Data fetched
```

31) How do you define an interface in TypeScript?
Answer:
An interface defines a contract for objects or classes specifying properties and types.
It ensures objects follow a consistent structure.
Code Example:
```
interface User {
id: number;
name: string;
age?: number; // optional
}
const user1: User = { id: 1, name: "Shrinivas" };
const user2: User = { id: 2, name: "Pratiksha", age: 30 };
```

32) What is the difference between interface and type alias?
Answer:
Interface can be extended or implemented and is generally preferred for objects and classes.
Type alias can define primitive, union, tuple, or object types, but cannot be implemented.
Code Example:
```
interface Person { name: string; }
interface Employee extends Person { id: number; }
type User = { name: string; id: number; };
type Admin = User & { role: string; };
```

33) How do you implement an interface in a class?
Answer:
A class implements an interface using the implements keyword and must define all properties/methods.
Code Example:
```
interface IUser {
id: number;
name: string;
getDetails(): string;
}
class User implements IUser {
constructor(public id: number, public name: string) {}
getDetails(): string {
return `${this.name} - ${this.id}`;
}
}
const u = new User(1, "Shrinivas");
console.log(u.getDetails()); // Shrinivas – 1
```

34) What is a type alias in TypeScript?
Answer:
Type alias gives a name to a type (primitive, object, union, or tuple) for reuse and readability.
Code Example:
```
type ID = number | string;
let userId: ID = 101;
userId = "A101"; // ✓ allowed
```

35) What are optional properties in interfaces?
Answer:
Optional properties are marked with ? and may or may not be present in the object.
Code Example:
```
interface Profile {
name: string;
age?: number;
}
const p1: Profile = { name: "Shrinivas" };
const p2: Profile = { name: "Pratik", age: 5 };
```

36) What is a readonly property in an interface or type?
Answer:
Readonly properties cannot be changed after initialization and are defined using the readonly keyword.
Code Example:
interface User {
readonly id: number;
name: string;
}
const user: User = { id: 1, name: "Shrinivas" };
// user.id = 2; ✗ error
user.name = "Pratiksha"; // ✓ allowed

37) What is an enum in TypeScript?
Answer:
An enum is a collection of named constants. It can be numeric or string-based.
Code Example:
enum Role {
Admin,
User,
Guest
}
let r: Role = Role.User;
console.log(r); // 1

38) What is a string enum?
Answer:
String enums assign custom string values to enum members instead of numeric.
Code Example:
enum Status {
Success = "SUCCESS",
Fail = "FAIL",
Pending = "PENDING"
}
let s: Status = Status.Success;
console.log(s); // SUCCESS

39) What are generics in TypeScript?
Answer:
Generics allow you to write reusable code for multiple types without losing type safety.
Code Example:
function identity<T>(value: T): T {
return value;
}
console.log(identity<number>(10)); // 10
console.log(identity<string>("Hello")); // Hello

40) How do you create a generic interface?
Answer:
Generics in interfaces make property types flexible while maintaining type safety.
Code Example:
interface ApiResponse<T> {
data: T;
success: boolean;
}
const response1: ApiResponse<string> = { data: "OK", success: true };
const response2: ApiResponse<number[]> = { data: [1, 2, 3], success: true };

41) Can interfaces extend classes?
Answer:
Yes, interfaces can extend classes.
The interface inherits all the class's public and protected members, but not private members.
Code Example:
class Person {
constructor(public name: string, protected age: number) {}
}
interface Employee extends Person {
employeeId: number;
}
const emp: Employee = { name: "Shrinivas", age: 40, employeeId: 101 };

42) Can interfaces be merged?
Answer:
Yes, interfaces with the same name are automatically merged by TypeScript.
This allows extending existing interfaces without modifying the original.
Code Example:
interface User {
id: number;
}
interface User {
name: string;
}
const u: User = { id: 1, name: "Shrinivas" }; // ✓ merged

43) What is optional chaining in interface?
Answer:
Optional chaining (?.) allows safely accessing nested or optional properties without throwing errors if they are undefined.
Code Example:
interface Profile {
name: string;
address?: {
city: string;
zip?: number;
};
}
const profile: Profile = { name: "Shrinivas" };
console.log(profile.address?.city); // undefined, no error

44) What is readonly in interface?
Answer:
Readonly properties cannot be changed after initialization.
They are defined using the readonly keyword.
Code Example:
interface User {
readonly id: number;
name: string;
}
const user: User = { id: 1, name: "Shrinivas" };
// user.id = 2; ✗ error
user.name = "Pratiksha"; // ✓ allowed

45) Can you use interface for function definition?
Answer:
Yes, interfaces can define the signature of a function, specifying parameter types and return type.
Code Example:
interface MathOperation {
(a: number, b: number): number;
}
const add: MathOperation = (x, y) => x + y;
console.log(add(5, 10)); // 15

46) Can interfaces define index signatures?
Answer:
Yes, index signatures allow interfaces to describe objects with dynamic property names.
Code Example:
interface StringDictionary {
[key: string]: string;
}
const colors: StringDictionary = {
primary: "red",
secondary: "blue"
};

47) When should you prefer interface over type?
Answer:
Prefer interface when defining object shapes or classes, because they can:
Be implemented by classes
Be extended
Be merged
Use type for unions, tuples, or primitives.
Code Example:
interface User { name: string; age: number; }
type ID = number | string;

```
const user: User = { name: "Shrinivas", age: 40 };
const id1: ID = 123;
const id2: ID = "A123";
```

48) What is a type alias?

Answer:

A type alias gives a name to a type (primitive, object, union, tuple, etc.) for reuse and readability.

Code Example:

```
type ID = number | string;
let userId: ID = 101;
userId = "A101"; // ✓ allowed
```

49) Difference between type alias and interface?

Answer:

Interface can be implemented by classes, extended, and merged.

Type alias is more flexible, can define primitives, unions, tuples, but cannot be implemented or merged.

Code Example:

```
interface User { name: string; }
type ID = number | string;
const u: User = { name: "Shrinivas" };
const id: ID = "A101";
```

50) Can type aliases use union types?

Answer:

Yes, type aliases can define union types using the | operator.

Code Example:

```
type Status = "success" | "fail" | "pending";
let s: Status;
s = "success"; // ✓
```

51) Can type aliases use intersection types?

Answer:

Yes, type aliases can combine multiple types using &.

Code Example:

```
type User = { name: string };
type Employee = { id: number };
type Admin = User & Employee;
const admin: Admin = { name: "Shrinivas", id: 101 };
```

52) Can type alias be extended?

Answer:

Type aliases cannot be implemented or merged, but you can extend them using intersection types (&).

Code Example:

```
type Person = { name: string };
type Employee = Person & { id: number };
const e: Employee = { name: "Shrinivas", id: 101 };
```

53) What is a mapped type?

Answer:

Mapped types allow you to create new types by transforming properties of an existing type.

Code Example:

```
type User = { name: string; age: number; };
type ReadonlyUser = { readonly [K in keyof User]: User[K] };
const u: ReadonlyUser = { name: "Shrinivas", age: 40 };
// u.name = "Pratik"; ✗ error
```

54) What is keyof?

Answer:

keyof extracts all the property names of a type as a union of string literals.

Code Example:

```
type User = { name: string; age: number; };
type UserKeys = keyof User; // "name" | "age"
const key: UserKeys = "name"; // ✓
```

55) What is typeof in TypeScript?

Answer:

typeof gets the type of a variable or object at compile time.

Code Example:

```
const user = { name: "Shrinivas", age: 40 };
type UserType = typeof user;
const u: UserType = { name: "Pratik", age: 5 }; // ✓
```

56) What is an indexed access type?
Answer:
Indexed access types let you get the type of a property from another type.
Code Example:
type User = { name: string; age: number; };
type NameType = User["name"]; // string
const n: NameType = "Shrinivas"; // ✅

57) What are conditional types?
Answer:
Conditional types select a type based on a condition using extends.
Code Example:
type Check<T> = T extends string ? "Yes" : "No";
type A = Check<string>; // "Yes"
type B = Check<number>; // "No"

58) How do you define a class in TypeScript?
Answer:
A class is a blueprint for creating objects.
It can contain properties, methods, and constructors.
Code Example:
class Person {
name: string;
age: number;
constructor(name: string, age: number) {
this.name = name;
this.age = age;
}
greet(): void {
console.log(`Hello, my name is ${this.name}`);
}
}
const p = new Person("Shrinivas", 40);
p.greet(); // Hello, my name is Shrinivas

59) What are access modifiers (public, private, protected)?
Answer:
Access modifiers control visibility of class members:
public – accessible everywhere (default)
private – accessible only inside the class
protected – accessible inside the class and subclasses
Code Example:
class Employee {
public name: string;
private salary: number;
protected department: string;
constructor(name: string, salary: number, department: string) {
this.name = name;
this.salary = salary;
this.department = department;
}
}

60) Difference between private and protected?
Answer:
private members are accessible only inside the class.
protected members are accessible inside the class and subclasses.
Code Example:
class Parent {
private privateProp = "private";
protected protectedProp = "protected";
}
class Child extends Parent {
showProps() {
// console.log(this.privateProp); ❌ error
console.log(this.protectedProp); // ✅ allowed
}
}

61) What is a constructor?
Answer:

A constructor is a special method called when an object is created.
It initializes class properties.
Code Example:
class User {
constructor(public name: string, public age: number) {}
}
const u = new User("Shrinivas", 40);
console.log(u.name, u.age); // Shrinivas 40

62) What is parameter properties?
Answer:
Parameter properties allow declaring and initializing class members directly in the constructor parameters using access modifiers.
Code Example:
class User {
constructor(public name: string, private age: number) {}
}
const u = new User("Shrinivas", 40);
console.log(u.name); // Shrinivas
// console.log(u.age); ✕ private

63) What is inheritance in TypeScript?
Answer:
Inheritance allows a class (child) to reuse properties and methods of another class (parent) using the extends keyword.
Code Example:
class Person {
constructor(public name: string) {}
greet() { console.log(`Hello, ${this.name}`); }
}
class Employee extends Person {
constructor(name: string, public id: number) {
super(name);
}
}
const e = new Employee("Shrinivas", 101);
e.greet(); // Hello, Shrinivas

64) What is method overriding?
Answer:
Method overriding occurs when a subclass provides its own implementation of a method from the parent class.
Code Example:
class Person {
greet() { console.log("Hello from Person"); }
}
class Employee extends Person {
greet() { console.log("Hello from Employee"); }
}
const e = new Employee();
e.greet(); // Hello from Employee

65) What is an abstract class?
Answer:
An abstract class cannot be instantiated directly and may contain abstract methods that must be implemented by subclasses.
Code Example:
abstract class Shape {
abstract area(): number;
describe(): void {
console.log("This is a shape");
}
}
class Circle extends Shape {
constructor(private radius: number) { super(); }
area(): number { return Math.PI * this.radius ** 2; }
}
const c = new Circle(5);
console.log(c.area());

66) Difference between abstract class and interface?
Answer:
Abstract class can contain implementation and properties, and supports inheritance.
Interface only defines structure/signatures and cannot have implementation (except optional properties with default values in TS 4.2+).
Code Example:
abstract class Animal { abstract makeSound(): void; }
interface IAnimal { makeSound(): void; }
class Dog extends Animal {
makeSound(): void { console.log("Woof"); }

```
}
class Cat implements IAnimal {
makeSound(): void { console.log("Meow"); }
}
```

67) Can a class implement multiple interfaces?
Answer:
Yes, a class can implement multiple interfaces, ensuring it provides all required members.
Code Example:

```
interface A { a(): void; }
interface B { b(): void; }
class C implements A, B {
a() { console.log("a"); }
b() { console.log("b"); }
}
const obj = new C();
obj.a(); // a
obj.b(); // b
```

68) What are generics?
Answer:
Generics allow you to write reusable and type-safe code that works with multiple types without losing type safety.
Code Example:

```
function identity<T>(value: T): T {
return value;
}
console.log(identity<number>(10)); // 10
console.log(identity<string>("Hello")); // Hello
```

69) Why are generics important in automation?
Answer:
Generics allow reusable components (like Page Objects, API response handlers, or utilities) to work with any type while preserving type safety, reducing runtime errors.
Code Example:

```
function wrapInArray<T>(item: T): T[] {
return [item];
}
const numbers = wrapInArray(10); // number[]
const strings = wrapInArray("Hello"); // string[]
```

70) How do you define a generic function?
Answer:
Generic functions define a type parameter using <T> and use it for parameters or return type.
Code Example:

```
function getFirst<T>(items: T[]): T {
return items[0];
}
console.log(getFirst<number>([1,2,3])); // 1
console.log(getFirst<string>(["a","b"])); // "a"
```

71) How do you define a generic interface?
Answer:
Generic interfaces use <T> to define flexible types for properties or methods.
Code Example:

```
interface ApiResponse<T> {
data: T;
success: boolean;
}
const response: ApiResponse<string> = { data: "OK", success: true };
const numberResponse: ApiResponse<number[]> = { data: [1,2,3], success: true };
```

72) How do you define a generic class?
Answer:
Generic classes allow the class to work with any type provided at instantiation.
Code Example:

```
class Box<T> {
constructor(public content: T) {}
getContent(): T { return this.content; }
}
const box1 = new Box<number>(123);
const box2 = new Box<string>("Hello");
console.log(box1.getContent()); // 123
console.log(box2.getContent()); // Hello
```

73) What is a generic constraint?
Answer:
Generic constraints restrict the types that can be passed to a generic type using extends.
Code Example:
function printName<T extends { name: string }>(obj: T) {
console.log(obj.name);
}
printName({ name: "Shrinivas", age: 40 }); // ✓

74) What is extends in generics?
Answer:
extends is used to restrict generic types to a certain shape or type.
Code Example:
interface HasLength { length: number; }
function logLength<T extends HasLength>(item: T) {
console.log(item.length);
}
logLength([1,2,3]); // 3
logLength("Hello"); // 5

75) What is default generic type?
Answer:
A default generic type provides a fallback type if none is specified.
Code Example:
function createArray<T = string>(items: T[]): T[] {
return items;
}
const arr1 = createArray(["a","b"]); // T inferred as string
const arr2 = createArray<number>([1,2,3]); // T explicitly number

76) Difference between generics and any?
Answer:
any disables type checking; you lose type safety.
generics preserve type safety while keeping code reusable.
Code Example:
function identityAny(value: any): any { return value; } // type safety lost
function identityGeneric<T>(value: T): T { return value; } // type safe

77) Real-time use case of generics in test frameworks?
Answer:
Generics are widely used in automation frameworks for Page Objects, API responses, data-driven testing, and utility functions.
They allow flexible yet type-safe code.
Code Example:
interface ApiResponse<T> {
data: T;
status: number;
}
async function fetchData<T>(url: string): Promise<ApiResponse<T>> {
const response = await fetch(url);
const data: T = await response.json();
return { data, status: 200 };
}
// Usage in test
interface User { id: number; name: string; }
fetchData<User[]>("/api/users").then(res => {
console.log(res.data[0].name); // type safe access
});

78) What is enum?
Answer:
An enum is a collection of named constants.
It is used to represent a set of related values with meaningful names.
Code Example:
enum Role {
Admin,
User,
Guest
}
let r: Role = Role.User;
console.log(r); // 1

79) Difference between numeric enum and string enum?
Answer:
Numeric enum: Values are auto-incremented numbers (default starting at 0).
String enum: Values are explicitly set as strings.
Code Example:

```
// Numeric enum
enum Direction {
Up, // 0
Down, // 1
Left, // 2
Right // 3
}
// String enum
enum Status {
Success = "SUCCESS",
Fail = "FAIL",
Pending = "PENDING"
}
console.log(Direction.Up); // 0
console.log(Status.Success); // SUCCESS
```

80) What is const enum?
Answer:
const enum is inlined by the compiler at compile time for performance.
No object is created at runtime.
Code Example:

```
const enum Colors {
Red,
Green,
Blue
}
let c: Colors = Colors.Green;
console.log(c); // 1
```

81) When should enums be avoided?
Answer:
Avoid enums if tree-shaking is important (can increase bundle size).
Avoid if simple union of string literals suffices.
Avoid enums in performance-critical scenarios.
Code Example (Alternative better approach):

```
// Instead of enum
type Status = "SUCCESS" | "FAIL" | "PENDING";
let s: Status = "SUCCESS";
```

82) What is alternative to enums in TypeScript?
Answer:
Use union types of string literals or const objects instead of enums for better tree-shaking and readability.
Code Example:

```
// String literal union
type Direction = "UP" | "DOWN" | "LEFT" | "RIGHT";
const move: Direction = "UP";
// Const object alternative
const Roles = {
Admin: "ADMIN",
User: "USER",
Guest: "GUEST"
} as const;
type Role = typeof Roles[keyof typeof Roles];
const r: Role = Roles.Admin;
```

83) What is Partial<T>?
Answer:
Partial<T> makes all properties of a type optional. Useful when updating objects partially.
Code Example:

```
interface User {
id: number;
name: string;
age: number;
}
const updateUser: Partial<User> = { name: "Shrinivas" }; // ✓ allowed
```

84) What is Required<T>?
Answer:
Required<T> makes all properties of a type required, even if they were optional.
Code Example:
interface Profile {
name: string;
age?: number;
}
const p: Required<Profile> = { name: "Shrinivas", age: 40 }; // ✅ must include age

85) What is Readonly<T>?
Answer:
Readonly<T> makes all properties of a type immutable (cannot be reassigned).
Code Example:
interface User {
id: number;
name: string;
}
const user: Readonly<User> = { id: 1, name: "Shrinivas" };
// user.id = 2; ❌ error

86) What is Pick<T, K>?
Answer:
Pick<T, K> creates a type by selecting a subset of properties from another type.
Code Example:
interface User {
id: number;
name: string;
age: number;
}
type UserNameOnly = Pick<User, "name">;
const user: UserNameOnly = { name: "Shrinivas" };

87) What is Omit<T, K>?
Answer:
Omit<T, K> creates a type by excluding certain properties from another type.
Code Example:
interface User {
id: number;
name: string;
age: number;
}
type UserWithoutAge = Omit<User, "age">;
const user: UserWithoutAge = { id: 1, name: "Shrinivas" };

88) What is Record<K, T>?
Answer:
Record<K, T> creates an object type with keys of type K and values of type T.
Code Example:
type Role = "Admin" | "User";
type RolePermissions = Record<Role, string[]>;
const permissions: RolePermissions = {
Admin: ["read", "write", "delete"],
User: ["read"]
};

89) What are utility types?
Answer:
Utility types are built-in TypeScript types that help transform existing types.
Examples: Partial, Required, Readonly, Pick, Omit, Record, NonNullable, ReturnType, Parameters.
Code Example:
interface User { id: number; name?: string; age?: number; }
type OptionalUser = Partial<User>; // all properties optional
type RequiredUser = Required<User>; // all properties required

90) What is NonNullable<T>?
Answer:
NonNullable<T> removes null and undefined from a type.
Code Example:
type Name = string | null | undefined;

type NonNullName = NonNullable<Name>; // string
const name: NonNullName = "Shrinivas"; // ✓

91) What is ReturnType<T>?
Answer:
ReturnType<T> extracts the return type of a function type.
Code Example:
function getUser() {
return { id: 1, name: "Shrinivas" };
}
type UserType = ReturnType<typeof getUser>;
const user: UserType = { id: 1, name: "Pratiksha" };

92) What is Parameters<T>?
Answer:
Parameters<T> extracts the parameter types of a function as a tuple.
Code Example:
function login(username: string, password: string) {}
type LoginParams = Parameters<typeof login>;
const args: LoginParams = ["shree", "12345"];

93) How does TypeScript help prevent runtime errors?
Answer:
TypeScript provides static type checking at compile time, ensuring variables, function parameters, and return values match expected types, reducing runtime errors.
Code Example:
function add(a: number, b: number): number {
return a + b;
}
// add("5", 10); ✗ compile-time error
console.log(add(5, 10)); // ✓ 15

94) What is type narrowing?
Answer:
Type narrowing is when TypeScript reduces a broad type to a more specific type using conditions or checks.
Code Example:
function printId(id: number | string) {
if (typeof id === "string") {
console.log(id.toUpperCase()); // narrowed to string
} else {
console.log(id); // narrowed to number
}
}

95) What is a type guard?
Answer:
A type guard is a runtime check that ensures a variable is of a specific type, allowing safe operations on it.
Code Example:
function isString(value: any): value is string {
return typeof value === "string";
}
function printValue(val: string | number) {
if (isString(val)) {
console.log(val.toUpperCase()); // ✓ safe
} else {
console.log(val); // number case
}
}

96) How do you write a custom type guard?
Answer:
Custom type guards use a function returning param is Type to narrow types.
Code Example:
interface User { name: string; }
interface Admin { role: string; }
function isAdmin(obj: User | Admin): obj is Admin {
return (obj as Admin).role !== undefined;
}
const a: Admin | User = { role: "ADMIN" };
if (isAdmin(a)) {
console.log(a.role); // ✓ safe
}

97) What is optional chaining (?.)?
Answer:
Optional chaining allows safe access to nested properties without throwing errors if a property is null or undefined.
Code Example:
interface Profile { name: string; address?: { city: string } }
const profile: Profile = { name: "Shrinivas" };
console.log(profile.address?.city); // undefined, no error

98) What is nullish coalescing (??)?
Answer:
?? returns the right-hand value only if the left-hand is null or undefined.
Code Example:
const name = null ?? "Default Name"; // Default Name
const age = 0 ?? 18; // 0 (because 0 is not null/undefined)

99) Difference between || and ???
Answer:
|| returns the right-hand value if the left-hand is falsy (0, "", false, null, undefined).
?? returns the right-hand value only if the left-hand is null or undefined.
Code Example:
console.log(0 || 10); // 10
console.log(0 ?? 10); // 0

100) What is never used for in exhaustive checks?
Answer:
never represents values that never occur and is useful in exhaustive type checks to ensure all cases are handled.
Code Example:
type Shape = { kind: "circle"; radius: number } | { kind: "square"; side: number };
function area(shape: Shape): number {
switch (shape.kind) {
case "circle": return Math.PI * shape.radius ** 2;
case "square": return shape.side ** 2;
default:
const _exhaustiveCheck: never = shape; // compile-time error if not handled
return _exhaustiveCheck;
}
}

101) How do you handle errors in async functions?
Answer:
Use try-catch blocks or .catch() with Promises to handle errors safely.
Code Example:
async function fetchData(url: string): Promise<string> {
try {
const res = await fetch(url);
if (!res.ok) throw new Error("Failed to fetch");
return await res.text();
} catch (error) {
console.error("Error:", error);
return "default data";
}
}

102) Why is unknown safer than any?
Answer:
unknown requires type checking before usage, unlike any which bypasses type safety, preventing runtime errors.
Code Example:
let value: unknown;
value = 10;
value = "Hello";
// console.log(value.toUpperCase()); ✗ error
if (typeof value === "string") {
console.log(value.toUpperCase()); // ✓ safe

103) What are ES modules?
Answer:
ES modules (ECMAScript modules) are a standardized way to export and import code between files.
They use export and import keywords.
Code Example:
// utils.ts

```
export function add(a: number, b: number): number {
return a + b;
}
// main.ts
import { add } from "./utils";
console.log(add(5, 10)); // 15
```

104) Difference between import and require?
Answer:
import is ES module syntax, supports static analysis, tree-shaking, and TypeScript types.
require is CommonJS syntax, used in Node.js.
Code Example:

```
// ES Module
import fs from "fs";
// CommonJS
const fs = require("fs");
```

105) What is tsconfig.json?
Answer:
tsconfig.json is the TypeScript project configuration file.
It specifies compiler options, files, include/exclude paths, and module settings.
Code Example:

```
{
"compilerOptions": {
"target": "ES6",
"module": "commonjs",
"strict": true
},
"include": ["src/**/*"]
}
```

106) Important compiler options for automation projects?
Answer:
target – JavaScript version output (ES5, ES6, etc.)
module – module system (commonjs, esnext)
strict – enable all strict type checks
esModuleInterop – enable default imports from CommonJS
baseUrl and paths – module path mapping
outDir – output directory
skipLibCheck – skip type checking for libraries

107) What is target and module?
Answer:
target specifies the JavaScript version to compile to.
module specifies the module system used in the compiled JS.
Code Example:

```
{
"compilerOptions": {
"target": "ES6",
"module": "commonjs"
}
}
```

108) What is baseUrl and paths?
Answer:
baseUrl sets the base directory for module resolution.
paths maps module aliases to paths for cleaner imports.
Code Example:

```
{
"compilerOptions": {
"baseUrl": "./src",
"paths": {
"@pages/*": ["pages/*"],
"@utils/*": ["utils/*"]
}
}
}
```

Usage in TS:

```
import { add } from "@utils/math";
```

109) What is types in tsconfig?
Answer:
types specifies type declaration packages to include globally.
Useful for test frameworks like Playwright, Jest, or Node types.

Code Example:
{
"compilerOptions": {
"types": ["node", "jest", "playwright"]
}
}

110) Difference between esModuleInterop true/false?
Answer:
true: Allows default import of CommonJS modules (import fs from "fs").
false: Requires namespace import (import * as fs from "fs").
Code Example:
// esModuleInterop: true
import fs from "fs";
// esModuleInterop: false
import * as fs from "fs";

111) What is skipLibCheck?
Answer:
skipLibCheck: true skips type checking of declaration files (.d.ts), speeding up compilation and avoiding errors from third-party libraries.
Code Example:
{
"compilerOptions": {
"skipLibCheck": true
}
}

112) What is a declaration file (.d.ts)?
Answer:
A .d.ts file declares types for JavaScript modules so TypeScript can understand them without implementation.
Used for third-party JS libraries or type definitions.
Code Example:
// lodash.d.ts
declare module "lodash" {
export function chunk<T>(array: T[], size?: number): T[][];
}
// usage in TS
import { chunk } from "lodash";
const result = chunk([1,2,3,4], 2); // [[1,2],[3,4]]

113) Why TypeScript is preferred in Playwright automation?
Answer:
TypeScript ensures type safety, better autocompletion, and early error detection.
It reduces runtime errors when interacting with locators, test data, or API responses.
Code Example:
import { Page } from "@playwright/test";
async function login(page: Page, email: string, password: string) {
await page.fill("#email", email);
await page.fill("#password", password);
await page.click("#submit");
}

114) How TypeScript improves maintainability in test frameworks?
Answer:
Type-safe Page Objects
Typed test data
Reusable utility functions
Easier refactoring and IDE support
Code Example:
interface User { email: string; password: string }
const testUser: User = { email: "test@example.com", password: "12345" };

115) How do you type Page Objects in TS?
Answer:
Use class with typed Page and locator properties.
Code Example:
import { Page, Locator } from "@playwright/test";
class LoginPage {
readonly email: Locator;
readonly password: Locator;
readonly submit: Locator;
constructor(private page: Page) {
this.email = page.locator("#email");
this.password = page.locator("#password");

```
this.submit = page.locator("#submit");
}
async login(email: string, password: string) {
await this.email.fill(email);
await this.password.fill(password);
await this.submit.click();
}
}
```

116) How do you type test data (JSON)?
Answer:
Use interfaces or types to define structure of test data.
Code Example:

```
interface UserData { name: string; age: number; }
const user: UserData = { name: "Shrinivas", age: 40 };
// Import JSON with type
import users from "./users.json" assert { type: "json" };
const firstUser: UserData = users[0];
```

117) How do you type API responses?
Answer:
Use interfaces or generics to define response types for type safety.
Code Example:

```
interface User { id: number; name: string }
async function fetchUsers(): Promise<User[]> {
const res = await fetch("/users");
return res.json() as Promise<User[]>;
}
```

118) How do you handle dynamic UI elements with strict typing?
Answer:
Use Locator type from Playwright and helper functions for dynamic selectors.
Code Example:

```
function getButton(page: Page, name: string): Locator {
return page.locator(`button:text("${name}")`);
}
await getButton(page, "Submit").click();
```

119) How do you avoid any in large frameworks?
Answer:
Use interfaces/types
Use generics for reusable utilities
Avoid as any and enable strict mode
Code Example:

```
function identity<T>(value: T): T { return value; } // generic instead of any
```

120) How do you share types across tests?
Answer:
Place interfaces/types in a separate folder, e.g., types/
Import wherever needed
Code Example:

```
// types/user.ts
export interface User { email: string; password: string }
// test.ts
import { User } from "../types/user";
const user: User = { email: "test@test.com", password: "123" };
```

121) How do you structure types folder?
Answer:
Suggested structure for large automation frameworks:

```
types/
├─ api/
│ └─ user.ts
├─ pages/
│ └─ loginPage.ts
├─ testData/
│ └─ users.ts
```

122) Common TypeScript mistakes in automation projects?
Answer:
Using any frequently
Ignoring strict mode
Not typing locators or API responses
Overusing type assertions (as)

Mixing ES module and CommonJS imports

123) Difference between interface merging and type redeclaration?
Answer:
Interface merging works automatically; multiple interfaces with same name are combined.
Type redeclaration fails, cannot declare same type twice.
Code Example:
interface User { id: number; }
interface User { name: string; } // ✓ merged
type ID = number;
// type ID = string; ✗ error

124) How does structural typing work?
Answer:
TypeScript uses duck typing; objects are compatible if they have the same shape, not necessarily the same name.
Code Example:
interface User { name: string }
const obj = { name: "Shrinivas", age: 40 };
const user: User = obj; // ✓ shape matches

125) What is duck typing?
Answer:
If an object has all required properties/methods, it is considered compatible with a type, regardless of actual type name.
Code Example:
interface Flyable { fly(): void }
const bird = { fly: () => console.log("flying") };
const airplane: Flyable = bird; // ✓ duck typing

126) How TypeScript compiles to JavaScript?
Answer:
TS is transpiled to JS by the TypeScript compiler.
All types are erased at runtime; only JS code remains.
Code Example:
// TypeScript
let x: number = 5;
// Compiles to JS
let x = 5;

127) Can TypeScript enforce runtime checks?
Answer:
No, TypeScript types exist only at compile time.
For runtime validation, use libraries like zod or io-ts.
Code Example:
import { z } from "zod";
const userSchema = z.object({ name: z.string(), age: z.number() });
userSchema.parse({ name: "Shrinivas", age: 40 }); // runtime validation

128) Why TypeScript types disappear at runtime?
Answer:
TypeScript is erased during compilation to JS to ensure compatibility with browsers and Node.js.
Types only exist for development and compile-time checks.
129) How do you debug TypeScript errors?
Answer:
Use IDE type hints (VS Code)
Enable "strict": true
Check type inference
Use tsc --noEmit to see compile errors

130) What happens if TS type mismatches runtime value?
Answer:
TypeScript cannot catch runtime mismatches
Can lead to runtime errors if JS receives unexpected type
Code Example:
let x: number = 5;
// JS runtime: x = "hello"; ✗ TypeScript won't prevent assignment in JS at runtime

131) How to write reusable strongly typed utilities?
Answer:
Use generics, interfaces, and utility types for flexibility and type safety.
Code Example:
function wrapInArray<T>(item: T): T[] { return [item]; }

```
const numbers = wrapInArray(5); // number[]
const strings = wrapInArray("hi"); // string[]
```

132) What is best TypeScript practice in test automation?
Answer:
Use strict mode
Avoid any
Use interfaces/types for Page Objects, API responses, and test data
Use generics for reusable utilities
Keep types in a central folder
Enable ES module imports for clarity
Validate runtime data with zod/io-ts if needed

1) Reverse a string

```
function reverseString(str: string): string {
return str.split("").reverse().join("");
}
console.log(reverseString("Shrinivas")); // "savinirhS"
```

2) Check if a string is a palindrome

```
function isPalindrome(str: string): boolean {
const reversed = str.split("").reverse().join("");
return str === reversed;
}
console.log(isPalindrome("madam")); // true
console.log(isPalindrome("Shrinivas")); // false
```

3) Count vowels in a string

```
function countVowels(str: string): number {
return (str.match(/[aeiouAEIOU]/g) || []).length;
}
console.log(countVowels("Shrinivas")); // 3
```

4) Remove duplicate characters from a string

```
function removeDuplicates(str: string): string {
return Array.from(new Set(str)).join("");
}
console.log(removeDuplicates("abbccddeff")); // "abcdef"
```

5) Find the first non-repeating character

```
function firstNonRepeating(str: string): string | null {
for (let char of str) {
if (str.indexOf(char) === str.lastIndexOf(char)) return char;
}
return null;
}
console.log(firstNonRepeating("aabbcdeeff")); // "c"
```

6) Capitalize the first letter of each word

```
function capitalizeWords(str: string): string {
return str
.split(" ")
.map(word => word.charAt(0).toUpperCase() + word.slice(1))
.join(" ");
}
console.log(capitalizeWords("hello world from ts")); // "Hello World From Ts"
```

7) Check if two strings are anagrams

```
function isAnagram(str1: string, str2: string): boolean {
const normalize = (s: string) => s.replace(/\s+/g, "").toLowerCase().split("").sort().join("");
return normalize(str1) === normalize(str2);
}
console.log(isAnagram("listen", "silent")); // true
console.log(isAnagram("hello", "world")); // false
```

8) Count occurrences of a character

```
function countChar(str: string, char: string): number {
return str.split("").filter(c => c === char).length;
}
```

```
console.log(countChar("Shrinivas", "s")); // 2
```

9) Reverse words in a sentence

```
function reverseWords(sentence: string): string {
return sentence.split(" ").reverse().join(" ");
}
console.log(reverseWords("hello world from typescript")); // "typescript from world hello"
```

10) Convert string to title case

```
function titleCase(str: string): string {
return str.toLowerCase().replace(/\b\w/g, char => char.toUpperCase());
}
console.log(titleCase("hello world from typescript")); // "Hello World From Typescript"
```

11) Check if string contains only digits

```
function isNumeric(str: string): boolean {
return /^\d+$/.test(str);
}
console.log(isNumeric("12345")); // true
console.log(isNumeric("123abc")); // false
```

12) Longest substring without repeating characters

```
function longestUniqueSubstring(str: string): string {
let maxSub = "", current = "";
for (let char of str) {
const index = current.indexOf(char);
if (index !== -1) current = current.slice(index + 1);
current += char;
if (current.length > maxSub.length) maxSub = current;
}
return maxSub;
}
console.log(longestUniqueSubstring("abcabcbb")); // "abc"
```

1. Basic Array Programs

1) Find largest number in array

```
function largest(arr: number[]): number {
return Math.max(...arr);
}
console.log(largest([1, 5, 3, 9])); // 9
```

2) Find smallest number in array

```
function smallest(arr: number[]): number {
return Math.min(...arr);
}
console.log(smallest([1, 5, 3, 9])); // 1
```

3) Sum of all elements

```
function sum(arr: number[]): number {
return arr.reduce((a, b) => a + b, 0);
}
console.log(sum([1, 2, 3, 4])); // 10
```

4) Average of array elements

```
function average(arr: number[]): number {
return arr.reduce((a, b) => a + b, 0) / arr.length;
}
console.log(average([1, 2, 3, 4])); // 2.5
```

5) Reverse an array

```
function reverseArray<T>(arr: T[]): T[] {
return arr.reverse();
}
console.log(reverseArray([1, 2, 3])); // [3,2,1]
```

6) Find second largest number

```
function secondLargest(arr: number[]): number {
```

```
const sorted = [...new Set(arr)].sort((a, b) => b - a);
return sorted[1];
}
console.log(secondLargest([1, 3, 2, 5, 5])); // 3
```

2. Array Searching & Filtering

7) Find even numbers

```
function evenNumbers(arr: number[]): number[] {
return arr.filter(n => n % 2 === 0);
}
console.log(evenNumbers([1, 2, 3, 4])); // [2,4]
```

8) Find odd numbers

```
function oddNumbers(arr: number[]): number[] {
return arr.filter(n => n % 2 !== 0);
}
console.log(oddNumbers([1, 2, 3, 4])); // [1,3]
```

9) Find duplicates

```
function duplicates<T>(arr: T[]): T[] {
return arr.filter((item, index) => arr.indexOf(item) !== index);
}
console.log(duplicates([1, 2, 2, 3, 3, 4])); // [2,3]
```

10) Remove duplicates

```
function removeDuplicates<T>(arr: T[]): T[] {
return Array.from(new Set(arr));
}
console.log(removeDuplicates([1, 2, 2, 3])); // [1,2,3]
```

3. Array Searching & Indexing

11) Find index of an element

```
function findIndex<T>(arr: T[], value: T): number {
return arr.indexOf(value);
}
console.log(findIndex([1, 2, 3], 2)); // 1
```

12) Find all indexes of a value

```
function allIndexes<T>(arr: T[], value: T): number[] {
return arr.map((v, i) => v === value ? i : -1).filter(i => i !== -1);
}
console.log(allIndexes([1,2,3,2,4], 2)); // [1,3]
```

13) Check if array includes a value

```
function includesValue<T>(arr: T[], value: T): boolean {
return arr.includes(value);
}
console.log(includesValue([1,2,3], 2)); // true
```

4. Sorting & Transformation

14) Sort numbers ascending/descending

```
function sortAsc(arr: number[]): number[] { return arr.sort((a,b)=>a-b); }
function sortDesc(arr: number[]): number[] { return arr.sort((a,b)=>b-a); }
console.log(sortAsc([3,1,2])); // [1,2,3]
console.log(sortDesc([3,1,2])); // [3,2,1]
```

15) Sort strings alphabetically

```
function sortStrings(arr: string[]): string[] {
return arr.sort();
}
console.log(sortStrings(["Banana","Apple","Mango"])); // ["Apple","Banana","Mango"]
```

16) Convert array to string (join)

```
const fruits = ["Apple", "Mango"];
console.log(fruits.join(", ")); // "Apple, Mango"
```

17) Convert string to array (split)

```
const str = "a,b,c";
const arr = str.split(",");
console.log(arr); // ["a","b","c"]
```

5. Array Math & Statistics

18) Find max & min without Math library

```
function maxMin(arr: number[]): [number, number] {
let max = arr[0], min = arr[0];
for(let num of arr) {
```

```
if(num>max) max=num;
if(num<min) min=num;
}
return [max, min];
}
console.log(maxMin([1,5,3])); // [5,1]
```

19) Sum of squares

```
function sumSquares(arr: number[]): number {
return arr.reduce((acc,n)=>acc+n*n,0);
}
console.log(sumSquares([1,2,3])); // 14
```

20) Average of numbers

```
function average(arr: number[]): number {
return arr.reduce((a,b)=>a+b,0)/arr.length;
}
console.log(average([1,2,3])); // 2
```

6. Advanced Array Programs

21) Merge two arrays and remove duplicates

```
function mergeArrays<T>(arr1: T[], arr2: T[]): T[] {
return Array.from(new Set([...arr1, ...arr2]));
}
console.log(mergeArrays([1,2,3],[2,3,4])); // [1,2,3,4]
```

22) Flatten nested arrays

```
function flattenArray<T>(arr: T[][]): T[] {
return arr.flat();
}
console.log(flattenArray([[1,2],[3,4]])); // [1,2,3,4]
```

23) Find all pairs with sum = X

```
function findPairs(arr: number[], sum: number): [number,number][] {
const result: [number,number][] = [];
for(let i=0;i<arr.length;i++){
for(let j=i+1;j<arr.length;j++){
if(arr[i]+arr[j]===sum) result.push([arr[i],arr[j]]);
}
}
return result;
}
console.log(findPairs([1,2,3,4],5)); // [[1,4],[2,3]]
```

24) Rotate array by N positions

```
function rotateArray<T>(arr: T[], n: number): T[] {
n = n % arr.length;
return [...arr.slice(n), ...arr.slice(0,n)];
}
console.log(rotateArray([1,2,3,4,5],2)); // [3,4,5,1,2]
```

25) Check if array is subset of another

```
function isSubset<T>(arr1: T[], arr2: T[]): boolean {
return arr1.every(val => arr2.includes(val));
}
console.log(isSubset([1,2],[1,2,3])); // true
console.log(isSubset([1,4],[1,2,3])); // false
```

7. Playwright + TypeScript Array Scenarios

26) Get all text from a list of elements

```
import { Page } from "@playwright/test";
async function getAllTexts(page: Page, selector: string): Promise<string[]> {
return page.locator(selector).allTextContents();
}
```

27) Filter buttons by text

```
import { Page } from "@playwright/test";
async function clickButtonByText(page: Page, text: string) {
const buttons = await page.locator("button").all();
for(let btn of buttons){
if(await btn.textContent() === text) await btn.click();
}
}
```

28) Map list of elements to their href

```
import { Page } from "@playwright/test";
async function getAllLinks(page: Page): Promise<string[]> {
return await page.locator("a").evaluateAll(els => els.map(a => a.getAttribute("href")));
```

```
}
```

1. Basic Number Programs

1) Check if a number is prime

```
function isPrime(num: number): boolean {
if(num < 2) return false;
for(let i = 2; i <= Math.sqrt(num); i++){
if(num % i === 0) return false;
}
return true;
}
console.log(isPrime(7)); // true
console.log(isPrime(10)); // false
```

2) Factorial of a number

```
function factorial(n: number): number {
if(n === 0) return 1;
return n * factorial(n-1);
}
console.log(factorial(5)); // 120
```

3) Fibonacci sequence (nth number)

```
function fibonacci(n: number): number {
if(n <= 1) return n;
return fibonacci(n-1) + fibonacci(n-2);
}
console.log(fibonacci(7)); // 13
```

4) Check if a number is even or odd

```
function isEven(n: number): boolean { return n % 2 === 0; }
function isOdd(n: number): boolean { return n % 2 !== 0; }
console.log(isEven(4)); // true
console.log(isOdd(5)); // true
```

5) Reverse a number

```
function reverseNumber(n: number): number {
const reversed = n.toString().split("").reverse().join("");
return parseInt(reversed);
}
console.log(reverseNumber(12345)); // 54321
```

2. Number Analysis Programs

6) Check if a number is palindrome

```
function isPalindromeNumber(n: number): boolean {
return n === reverseNumber(n);
}
console.log(isPalindromeNumber(121)); // true
console.log(isPalindromeNumber(123)); // false
```

7) Count digits in a number

```
function countDigits(n: number): number {
return n.toString().length;
}
console.log(countDigits(12345)); // 5
```

8) Sum of digits of a number

```
function sumDigits(n: number): number {
return n.toString().split("").reduce((a,b)=>a+parseInt(b),0);
}
console.log(sumDigits(1234)); // 10
```

9) Check Armstrong number

```
function isArmstrong(n: number): boolean {
const digits = n.toString().split("").map(Number);
const sum = digits.reduce((a,b)=>a+Math.pow(b,digits.length),0);
return sum === n;
}
console.log(isArmstrong(153)); // true
console.log(isArmstrong(123)); // false
```

10) Greatest common divisor (GCD)

```
function gcd(a: number, b: number): number {
while(b) {
[a, b] = [b, a % b];
}
```

```
return a;
}
console.log(gcd(12, 18)); // 6
```

11) Least common multiple (LCM)

```
function lcm(a: number, b: number): number {
return (a * b) / gcd(a,b);
}
console.log(lcm(12, 18)); // 36
```

3. Number Transformations

12) Convert decimal to binary

```
function decimalToBinary(n: number): string {
return n.toString(2);
}
console.log(decimalToBinary(10)); // "1010"
```

13) Convert binary to decimal

```
function binaryToDecimal(bin: string): number {
return parseInt(bin, 2);
}
console.log(binaryToDecimal("1010")); // 10
```

14) Convert decimal to hex

```
function decimalToHex(n: number): string {
return n.toString(16);
}
console.log(decimalToHex(255)); // "ff"
```

15) Round a number to n decimal places

```
function roundTo(n: number, digits: number): number {
return parseFloat(n.toFixed(digits));
}
console.log(roundTo(3.14159, 2)); // 3.14
```

4. Advanced Number Programs

16) Check if a number is perfect

```
function isPerfect(n: number): boolean {
let sum = 0;
for(let i=1;i<n;i++){
if(n % i === 0) sum += i;
}
return sum === n;
}
console.log(isPerfect(6)); // true
console.log(isPerfect(28)); // true
```

17) Find factorial using loop

```
function factorialLoop(n: number): number {
let fact = 1;
for(let i=1;i<=n;i++) fact *= i;
return fact;
}
console.log(factorialLoop(5)); // 120
```

18) Generate first N Fibonacci numbers

```
function fibonacciSeries(n: number): number[] {
const series: number[] = [0,1];
for(let i=2;i<n;i++) series[i] = series[i-1] + series[i-2];
return series.slice(0,n);
}
console.log(fibonacciSeries(7)); // [0,1,1,2,3,5,8]
```

19) Sum of all primes up to N

```
function sumPrimes(n: number): number {
let sum = 0;
for(let i=2;i<=n;i++){
if(isPrime(i)) sum += i;
}
return sum;
}
console.log(sumPrimes(10)); // 17 (2+3+5+7)
```

20) Count number of prime numbers in an array

```
function countPrimes(arr: number[]): number {
return arr.filter(isPrime).length;
}
```

```
console.log(countPrimes([1,2,3,4,5,6,7])); // 4
```

5. Playwright + TypeScript Number Scenarios

21) Count number of elements on page

```
import { Page } from "@playwright/test";
async function countElements(page: Page, selector: string): Promise<number> {
return await page.locator(selector).count();
}
```

22) Sum of numeric values from a table column

```
import { Page } from "@playwright/test";
async function sumColumn(page: Page, selector: string): Promise<number> {
const values = await page.locator(selector).allTextContents();
return values.map(Number).reduce((a,b)=>a+b,0);
}
```

23) Find max numeric value from table cells

```
import { Page } from "@playwright/test";
async function maxValue(page: Page, selector: string): Promise<number> {
const values = await page.locator(selector).allTextContents();
return Math.max(...values.map(Number));
}
```

6. Miscellaneous Number Programs

24) Check if a number is a power of 2

```
function isPowerOfTwo(n: number): boolean {
return n > 0 && (n & (n-1)) === 0;
}
console.log(isPowerOfTwo(16)); // true
console.log(isPowerOfTwo(18)); // false
```

25) Count trailing zeros in factorial of a number

```
function trailingZerosFactorial(n: number): number {
let count = 0;
for(let i=5;i<=n;i*=5) count += Math.floor(n/i);
return count;
}
console.log(trailingZerosFactorial(100)); // 24
```

26) Reverse digits of a number without converting to string

```
function reverseDigits(n: number): number {
let rev = 0;
while(n > 0){
rev = rev*10 + n%10;
n = Math.floor(n/10);
}
return rev;
}
console.log(reverseDigits(12345)); // 54321
```

27) Check if a number is a perfect square

```
function isPerfectSquare(n: number): boolean {
return Number.isInteger(Math.sqrt(n));
}
console.log(isPerfectSquare(16)); // true
console.log(isPerfectSquare(20)); // false
```