TypeScript

🧩 Session 1 — Type Building Blocks

🎯 Goals

By the end of this session, you'll be able to:

- Confidently use **primitive types** (`number`, `string`, `boolean`, etc.)
- Create **type aliases** and **interfaces**
- Understand **optional**, **readonly**, and **union** properties
- Write functions with clear **type annotations**

1️⃣ Primitives & Basic Typing

```
tsCopy codelet id: number = 101;
let name: string = "Fuzail";
let isActive: boolean = true;
let score: number | null = null; // union type
```

Key points

- **Type inference:** TypeScript infers types automatically — e.g. `let age = 20` → `number`.
- **Union types:** Use `|` when a variable can hold multiple types.
- **Any:** Avoid `any`; prefer specific types or generics.
- **Unknown:** Use `unknown` if type is truly uncertain and you'll check it later.

2️⃣ Type Aliases

A **type alias** creates a reusable name for a type or combination of types.

```
tsCopy codetype ID = number | string;
type Direction = "N" | "S" | "E" | "W";
```

Example:

```
tsCopy codefunction getIdAsString(id: ID): string {
  return String(id);
}
```

👉 **Use `type`** when combining unions, tuples, or mapped types.

3️⃣ Interfaces

Interfaces define the **shape** of an object.

They can include **readonly** or **optional** properties and **methods**.

```
tsCopy codeinterface Person {
  readonly id: number;
  name: string;
  age?: number; // optional
  greet(): string;
}

const user: Person = {
  id: 1,
  name: "Fuzail",
  greet() {
    return `Hi, I'm ${this.name}`;
  },
};
```

Notes:

- `readonly` prevents reassignment (`user.id = 2` ❌)
- `?` marks optional properties
- Interfaces can **extend** others for reuse

```
tsCopy codeinterface Employee extends Person {
  department: string;
}
```

4️⃣ Structural Typing & Excess Property Checks

TypeScript uses **structural typing**:

If two types have the same structure, they are compatible — even if declared separately.

```
tsCopy codeinterface Point { x: number; y: number }
interface Coord { x: number; y: number }
```

```
const p: Point = { x: 1, y: 2 };
const c: Coord = p; // ✅ shapes match
```

> But beware of excess property checks when directly assigning literals:

```
tsCopy codeconst rect: { width: number; height: number } = { width: 10, heigh
t: 5, color: 'red' };
// ❌ Error: 'color' does not exist in type
```

To bypass that, use an intermediate variable or `as`:

```
tsCopy codeconst temp = { width: 10, height: 5, color: 'red' };
const rect = temp; // ✅
```

🧠 Practice Examples

Example 1: Binary Tree Node

```
tsCopy codeinterface TreeNode {
  val: number;
  left: TreeNode | null;
  right: TreeNode | null;
}

function sumTree(root: TreeNode | null): number {
  if (!root) return 0;
  return root.val + sumTree(root.left) + sumTree(root.right);
}
```

Example 2: Direction movement

```
tsCopy codetype Direction = "N" | "S" | "E" | "W";

function move(directions: Direction[]): [number, number] {
  let x = 0, y = 0;
  for (const dir of directions) {
    if (dir === "N") y++;
    if (dir === "S") y--;
    if (dir === "E") x++;
    if (dir === "W") x--;
  }
  return [x, y];
}
```

🧩 Exercises (Your turn!)

Try writing these in your editor or TypeScript Playground

:

Exercise 1

Create an **interface** `TreeNode` with `val`, `left`, and `right`.

Then implement a function:

```
tsCopy codefunction countNodes(root: TreeNode | null): number;
```

Return how many nodes exist in the tree.

Exercise 2

Define a **type alias**:

```
tsCopy codetype Coordinate = [number, number];
```

Write a function:

```
tsCopy codefunction distance(a: Coordinate, b: Coordinate): number;
```

It should return the Euclidean distance.

Exercise 3

Create an **interface** `User` with:

- `id` (number)
- `username` (string)
- optional `email`
- method `getContactInfo(): string`

Then make two users and call `getContactInfo` for each.

Exercise 4

Explain (in your own words) why this assignment fails:

```
tsCopy codeinterface Car { make: string }
const car: Car = { make: "Tesla", color: "red" }; // ❌
```

and how to fix it.

🏡 Homework (mini LeetCode-style)

Implement **Two Sum** using proper typing.

```
tsCopy codefunction twoSum(nums: number[], target: number): [number, number] | null {
  // your code
}
```

✅ Hint: next session, you'll replace your object-based solution with a typed `Record` or `Map`.

Would you like me to:

1. Show **my sample solutions** for these exercises (so you can compare),

   or

2. Wait for **your code submissions** first so I can review and give feedback?

You said:

I will solve them later and then submit to you for check for now move to next phase if first is complete.

ChatGPT said:

Excellent — that's exactly the right mindset 👍

✅ **Session 1 complete.**

You now understand how to build and use custom types (`type`, `interface`, unions, optional/readonly props, structural typing).

🧭 Next Phase — **Session 2: Objects, Records & Index Signatures**

This session takes you one step deeper: learning how to build **dynamic key-value structures** — the ones used all the time in LeetCode problems like *Two Sum*, *Group Anagrams*, *Top K Frequent Elements*, etc.

🎯 Goals

By the end of this session, you'll:

- Know how to type **plain objects** used as dictionaries.
- Understand **index signatures** and the built-in `Record<K,V>` type.
- Learn when to use **objects** vs **Map** (next session).
- Implement frequency counters and typed dictionaries cleanly.

1️⃣ Index Signatures

Used when keys are **unknown at compile time** (like dynamic strings or numbers).

```
tsCopy codeinterface FrequencyCounter {
  [key: string]: number;
}

const freq: FrequencyCounter = {};
freq["apple"] = 3;
freq["banana"] = 1;
```

🧠 **Explanation:**

- `[key: string]: number` means *any* string key maps to a `number`.
- Every value must match the value type (`number` here).

⚠️ **Pitfall:** Numeric keys are automatically converted to strings at runtime!

```
tsCopy codeconst obj: FrequencyCounter = {};
obj[1] = 100;
console.log(Object.keys(obj)); // ["1"]
```

2️⃣ `Record<K, V>` — The modern shortcut

`Record` is a built-in TypeScript utility that expresses the same idea more succinctly:

```
tsCopy codeconst freq: Record<string, number> = {};
```

Equivalent to:

```
tsCopy codeinterface FrequencyCounter { [key: string]: number }
```

✅ It's generic:

```
tsCopy codetype StudentGrades = Record<string, number>;
```

✅ You can use union keys:

```
tsCopy codetype Directions = "N" | "S" | "E" | "W";
type Movement = Record<Directions, number>;

const moveCount: Movement = { N: 0, S: 0, E: 0, W: 0 };
```

3️⃣ Object vs Map — preview

Feature	Object	Map
Key types	string/symbol (numbers → string)	any value
Order	not guaranteed	preserves insertion order
Iteration	`for...in` or `Object.keys()`	`map.forEach()` or spread
Typical use	small frequency counters	large datasets, non-string keys

---

4️⃣ Practice Examples

Example 1 – Character Frequency Counter

```
tsCopy codefunction charCount(s: string): Record<string, number> {
  const count: Record<string, number> = {};
  for (const ch of s) {
    count[ch] = (count[ch] || 0) + 1;
  }
  return count;
}
```

Example 2 – Two Sum using a dictionary

```
tsCopy codefunction twoSum(nums: number[], target: number): [number, number] | null {
  const seen: Record<number, number> = {}; // number keys → coerced to string at runtime
  for (let i = 0; i < nums.length; i++) {
    const diff = target - nums[i];
    if (seen[diff] !== undefined) return [seen[diff], i];
    seen[nums[i]] = i;
  }
  return null;
}
```

🧩 Exercises

Exercise 1

Create an interface `WordCount` that maps words to their counts using an index signature.

Then implement:

```
tsCopy codefunction countWords(text: string): WordCount;
```

Split on spaces and count each unique word.

Exercise 2

Use `Record<string, boolean>` to track visited cities.

Implement:

```
tsCopy codefunction allVisited(cities: string[], visited: Record<string, boolean
>): boolean;
```

Return `true` if all cities appear in `visited`.

Exercise 3

Explain why `Record<number, string>` internally still stores keys as strings.

Write a small snippet showing that `"1"` and `1` refer to the same property.

Exercise 4

Build a simple **frequency map for an array of numbers**.

Then modify it to find the number with **maximum frequency**.

🏡 Homework

Re-implement your `Two Sum` function from Session 1 using a `Record<number, number>`.

✅ Try to spot where key coercion happens when you log the dictionary.

When you finish, you'll be ready for **Session 3 → "Map & Set"**, where we remove the string-key limitation entirely and use ES6 collections.

Would you like me to **continue to Session 3 now**,

or do you want to **pause here and practice Session 2 exercises first** before moving on?

You said:

Move to session 3 and you should add as much details to each as possible.

ChatGPT said:

Perfect — you're progressing just right 🚀

Let's now move on to **Session 3: Maps & Sets in TypeScript** — one of the most essential parts for solving **LeetCode problems** that rely on key–value pairs, lookups, and unique elements efficiently.

This session will go deep into:

- ES6 `Map` and `Set` data structures,
- Their typed equivalents in TypeScript,
- When to use them over plain objects,
- Advanced patterns with LeetCode-style problems,
- And plenty of detailed exercises to strengthen your grip.

🧩 Session 3 — `Map` & `Set` (Typed Collections)

🎯 Learning Goals

By the end of this session, you'll be able to:

1. Define and use typed `Map<K, V>` and `Set<T>` structures in TypeScript.
2. Understand **their methods, iteration patterns, and runtime differences** from plain objects.
3. Choose between **Map**, **Object**, and **Record** effectively depending on the problem.
4. Solve typical LeetCode-style problems like **Two Sum**, **Group Anagrams**, **Top K Frequent Elements**, and **Unique Numbers** using Map/Set cleanly.

🧱 1️⃣ What is a `Map`?

A **Map** is a collection of *key–value pairs* that remembers the original insertion order of the keys.

Unlike plain objects, **keys in Map can be of any type** — not just strings or symbols.

```
tsCopy codeconst map = new Map<string, number>();
map.set("apple", 5);
map.set("banana", 3);

console.log(map.get("apple")); // 5
console.log(map.has("banana")); // true
console.log(map.size); // 2
```

🧠 Why use Map instead of Object?

Feature	`Object`	`Map`
Key types	Only string or symbol	Any type (string, number, object, etc.)
Key order	Not guaranteed (ES spec doesn't rely on it)	Preserves insertion order

Feature	`Object`	`Map`
Iteration	`for...in`, `Object.keys()`	`for...of`, `map.forEach()`
Performance	Slower for frequent insert/delete	Optimized for frequent insert/delete
Default prototype keys	Inherited from `Object.prototype`	Pure key–value structure

👉 **Rule of thumb:**

- Use **Object / Record** → When keys are fixed, small, string-based.
- Use **Map** → When keys are dynamic, may be non-string, or frequent insertions/deletions occur.

🔢 2️⃣ Typed Map in TypeScript

Map is generic — you specify both key and value types:

```
tsCopy codeconst userAges = new Map<string, number>();
userAges.set("Fuzail", 21);
userAges.set("Ali", 20);
```

You can even use **objects** as keys:

```
tsCopy codeconst key1 = { id: 1 };
const key2 = { id: 2 };

const userMap = new Map<object, string>();
userMap.set(key1, "Fuzail");
userMap.set(key2, "Ali");

console.log(userMap.get(key1)); // "Fuzail"
```

🔎 Each key reference is **unique by reference**, not by content.

If you create another `{ id: 1 }`, it won't match the existing key.

🧭 3️⃣ Map Methods Cheat Sheet

Method	Description	Example
`.set(key, value)`	Add or update a value	`map.set('a', 1)`
`.get(key)`	Retrieve value by key	`map.get('a')`
`.has(key)`	Check if key exists	`map.has('a')`
`.delete(key)`	Remove a key-value pair	`map.delete('a')`
`.clear()`	Remove all pairs	`map.clear()`
`.size`	Get total entries	`map.size`
`.keys()`	Returns an iterator for keys	`for (const k of map.keys())`
`.values()`	Iterator for values	`for (const v of map.values())`
`.entries()`	Iterator for `[key, value]` pairs	`for (const [k,v] of map.entries())`
`.forEach()`	Executes callback for each element	`map.forEach((v,k)=>{})`

💡 4️⃣ Example 1 — Frequency Map

```
tsCopy codefunction frequencyMap(nums: number[]): Map<number, number
> {
  const map = new Map<number, number>();
  for (const num of nums) {
    map.set(num, (map.get(num) || 0) + 1);
  }
  return map;
}
```

✅ Why Map is better here:

- We might have large numeric ranges or negative numbers (no string coercion).

- Insertions and lookups are consistent and efficient.

💡 5️⃣ Example 2 — Two Sum (using Map)

```
tsCopy codefunction twoSum(nums: number[], target: number): [number, num
ber] | null {
  const map = new Map<number, number>();

  for (let i = 0; i < nums.length; i++) {
    const diff = target - nums[i];
    if (map.has(diff)) {
      return [map.get(diff)!, i];
    }
    map.set(nums[i], i);
  }
  return null;
}
```

🧠 **LeetCode Tip:**

This solution runs in **O(n)** time, **O(n)** space, and avoids string coercion issues that plain objects have.

🌿 6️⃣ Iterating Over Map

```
tsCopy codefor (const [key, value] of map) {
  console.log(key, value);
}
```

Or using destructuring and higher-order functions:

```
tsCopy code[...map.entries()].forEach(([key, value]) => console.log(key, valu
e));
```

🧩 7️⃣ What is a `Set`?

A **Set** is a collection of *unique* values — it automatically removes duplicates.

```
tsCopy codeconst uniqueNumbers = new Set<number>([1, 2, 2, 3, 4]);
console.log(uniqueNumbers); // Set(4) {1, 2, 3, 4}
```

🧠 Set Methods

Method	Description
`.add(value)`	Add a new element
`.delete(value)`	Remove an element
`.has(value)`	Check if exists
`.clear()`	Remove all elements
`.size`	Count of elements
`.values()`	Iterator of values

💡 Example — Detect Duplicates (LeetCode "Contains Duplicate")

```
tsCopy codefunction hasDuplicate(nums: number[]): boolean {
  const seen = new Set<number>();
  for (const n of nums) {
    if (seen.has(n)) return true;
    seen.add(n);
  }
  return false;
}
```

✅ O(n) time, O(n) space.

✅ Much simpler than using a `Record`.

⚔️ 8️⃣ Map vs Set Quick Summary

Use Case	Use
You need key–value lookups	`Map<K, V>`
You need to track unique values only	`Set<T>`
You want ordered key iteration	`Map`
You want quick membership check	`Set`
You want to count frequencies	`Map`

🧠 9️⃣ Advanced Example — Group Anagrams (Map + sorting keys)

```
tsCopy codefunction groupAnagrams(words: string[]): string[][] {
  const map = new Map<string, string[]>();

  for (const word of words) {
    const key = word.split("").sort().join(""); // canonical form
    if (!map.has(key)) map.set(key, []);
    map.get(key)!.push(word);
  }

  return Array.from(map.values());
}
```

✅ Time complexity: O(n * k log k)

✅ Space complexity: O(n * k)

🧩 1️⃣0️⃣ Exercises

Exercise 1

Implement a `Map` that tracks how many times each character appears in a string.

Return the character that appears most frequently.

Exercise 2

Write a function that takes two arrays and returns `true` if they share **any common element** using a `Set`.

```
tsCopy codefunction hasCommon(arr1: number[], arr2: number[]): boolean;
```

Exercise 3

Implement a **bidirectional map** using `Map<string, string>` where you can look up a value by key *or* a key by value.

Exercise 4

Given a list of student → course pairs, use a `Map<string, Set<string>>` to group all courses per student.

```
tsCopy codefunction groupCourses(pairs: [string, string][]): Map<string, Set<
string>>;
```

🏁 Session Summary

You now understand:

- How `Map<K,V>` provides strong typing and fast lookups.
- How `Set<T>` ensures uniqueness.
- When to prefer `Map` over `Object` / `Record` .
- Common LeetCode patterns using them.

Next up 👉 **Session 4: Tuples & Advanced Array Types**

We'll explore **tuple types**, **readonly tuples**, **destructuring with type safety**, and real-world LeetCode examples (like returning coordinates or [value, index] pairs).

Would you like me to move ahead to **Session 4 (Tuples & Advanced Array Types)** now,

or would you prefer I first add **some in-depth notes & visual memory aids (tables + diagrams)** for `Map` and `Set` before we continue?

...

You said:

Go for session 4

ChatGPT said:

Excellent 🎯 — you're doing this perfectly, step by step.

Now, we move into **Session 4: Tuples & Advanced Array Types**, one of the most useful and misunderstood areas in TypeScript.

Tuples often appear in **LeetCode problems** when returning structured pairs, coordinates, or multiple result values like `[index, value]`, `[row, col]`, `[node, level]`, etc.

Let's make this your strongest concept yet 💪

🧩 Session 4 — Tuples & Advanced Array Types

🎯 Learning Goals

By the end of this session, you'll be able to:

- Understand the **difference between arrays and tuples**.
- Define and use **typed tuples** (fixed-length, ordered data).
- Use **readonly** and **optional tuple elements**.
- Apply tuples to **LeetCode-style problems** (returning multiple values, coordinates, etc).
- Learn **advanced tuple operations** like destructuring, rest elements, and mapped tuples.

🧱 1️⃣ What is a Tuple?

A **tuple** is a fixed-length array with known types for each index.

```
tsCopy codeconst person: [string, number] = ["Fuzail", 21];
```

🧠 Explanation:

- At index 0 → string
- At index 1 → number
- TypeScript enforces both order and type.

If you try to reverse them:

```
tsCopy codeconst invalidPerson: [string, number] = [21, "Fuzail"]; // ❌ Error
```

So unlike arrays (which are flexible and homogenous), tuples are **structured, positional, and strongly typed**.

⚖️ 2️⃣ Array vs Tuple — Key Differences

Feature	Array	Tuple
Length	Flexible	Fixed (known at compile time)
Element types	Usually same	Each position can have different type
Index-based access	Allowed	Allowed, with specific type per index
Common use	Lists of similar items	Structured pair/group of related values

Example:

```
tsCopy codeconst numbers: number[] = [1, 2, 3];        // same type
const pair: [number, string] = [42, "key"];  // mixed, fixed length
```

🔧 3️⃣ Readonly Tuples

Tuples can be **immutable** — preventing accidental modification.

```
tsCopy codeconst point: readonly [number, number] = [10, 20];
// point[0] = 15; ❌ Error: cannot assign to readonly property
```

Useful for returning coordinates or fixed output from functions.

🔍 4️⃣ Optional Elements in Tuples

Sometimes a tuple can have optional parts:

```
tsCopy codetype PersonInfo = [string, number?, string?];

const p1: PersonInfo = ["Ali"];
const p2: PersonInfo = ["Fuzail", 21];
const p3: PersonInfo = ["Zain", 22, "Engineer"];
```

Optional elements must always be at the **end**.

🧠 5️⃣ Tuple Destructuring

You can unpack tuple values directly:

```
tsCopy codeconst user: [string, number] = ["Fuzail", 21];
const [name, age] = user;
console.log(`${name} is ${age} years old.`);
```

This is super common in LeetCode when you return `[index, value]` or `[row, col]` pairs.

💡 6️⃣ Example 1 — Returning Multiple Values

LeetCode-style example:

Problem: Find the first index where the sum of two elements equals target.

```
tsCopy codefunction findPair(nums: number[], target: number): [number, num
ber] | null {
  const map = new Map<number, number>();
  for (let i = 0; i < nums.length; i++) {
    const diff = target - nums[i];
    if (map.has(diff)) return [map.get(diff)!, i];
    map.set(nums[i], i);
  }
  return null;
}
```

The return type `[number, number] | null` is a **tuple type** that clearly defines the result's shape.

🧮 7️⃣ Example 2 — Matrix Coordinates

Let's say you're traversing a 2D grid:

```
tsCopy codetype Coordinate = [number, number];

function getNeighbors([row, col]: Coordinate): Coordinate[] {
  const directions: Coordinate[] = [
    [1, 0],
    [-1, 0],
    [0, 1],
    [0, -1],
  ];
  return directions.map(([dr, dc]) => [row + dr, col + dc]);
}
```

✅ Cleanly represents pairs of numbers (rows & columns).

✅ Easily destructurable.

✅ Very common in BFS/DFS problems.

🧠 8️⃣ Tuples with Rest Elements

You can allow variable-length tuples using rest syntax:

```
tsCopy codetype LogEntry = [string, ...number[]];

const logs: LogEntry = ["CPU", 10, 20, 30, 40];
```

First element is a string (category), followed by any number of numeric readings.

🧩 9️⃣ Advanced — Tuple Manipulation Types

TypeScript 4+ introduced **tuple type operators** for more complex transformations:

```
tsCopy codetype Pair = [string, number];
type Reverse<T extends any[]> = T extends [infer A, infer B] ? [B, A] : never;

type Swapped = Reverse<Pair>; // [number, string]
```

This concept (using `infer`) appears in **type-level programming** — helpful for writing reusable utilities.

🔥 1️⃣0️⃣ Real LeetCode-Like Use Cases

Problem	Tuple Role	Example Type
Two Sum	Return index pair	`[number, number]`
BFS / Graph	Store queue coordinates	`[number, number]`
Dijkstra / A*	Store `[node, distance]`	`[number, number]`
Custom Sorting	Return `[value, index]`	`[number, number]`

🧩 1️⃣1️⃣ Exercises

Exercise 1

Create a tuple type `StudentRecord` that stores:

- Name (string)
- Roll number (number)
- Grade (string)

  Then write a function `printStudent(record: StudentRecord): void` that prints it nicely.

Exercise 2

Create a function:

```
tsCopy codefunction swap<T, U>(pair: [T, U]): [U, T];
```

It should swap the two elements of a tuple and return the result.

Try calling `swap([1, "apple"])`.

Exercise 3

Write a function:

```
tsCopy codefunction processPoints(points: readonly [number, number][]): void;
```

It should log each coordinate pair using destructuring and prevent modification.

Exercise 4

Define a tuple type `Coordinate` and use it to write a BFS function that explores all valid neighboring cells in a grid of size `n × n`.

The function should print all visited coordinates.

Exercise 5 (Advanced)

Create a generic type `Last<T extends any[]>` that extracts the last element of a tuple type.

```
tsCopy codetype Last<[1, 2, 3]>  // → 3
```

Hint: Use conditional types with `infer`.

🧭 Quick Summary

Concept	Key Idea
Tuple	Fixed-length, position-based array
Optional Elements	`?` for flexibility
Readonly	Prevents modification
Rest elements	Allows variable length
Common use cases	Returning pairs, coordinates, structured results

🚀 Next Phase — **Session 5: Dictionaries, Records & Advanced Object Typing**

In the next session, we'll go deep into **typed key-value mappings**,

combining what you learned about tuples, objects, and maps to handle structured data cleanly.

It will cover:

- `Record<K, V>` advanced patterns,
- Dynamic key typing (`keyof`, indexed access),
- Nested and computed object types,
- And dictionary-like data structures in LeetCode problems like "Group Anagrams" or "Word Pattern".

Would you like me to continue straight to **Session 5 (Dictionaries, Records & Advanced Object Typing)**

or would you prefer that I first add **visual memory charts for tuples (comparison, anatomy, and patterns)** before we continue?

...

You said:

Yes move next.

ChatGPT said:

Perfect 🔥 — you're progressing steadily and deeply, just like a top problem solver should.

Welcome to the **next phase** in our TypeScript mastery series for LeetCode problem solving.

🧩 Session 5 — Dictionaries, Records & Advanced Object Typing

🎯 Learning Goals

By the end of this session, you'll be able to:

- Use `Record<K, V>` and **index signatures** for typed dictionaries.
- Dynamically create and access **object-based maps**.
- Use `keyof`, `typeof`, and **indexed access types**.
- Write **generic functions** that operate on objects safely.
- Handle LeetCode-style problems like frequency counting, grouping, and mapping efficiently.

🧱 1️⃣ The Problem with Plain Objects

You already know that objects in JavaScript are often used as maps:

```
tsCopy codeconst scores = {
  Ali: 90,
  Fuzail: 85,
};
```

✅ Simple

❌ But **not type-safe** if keys or values are dynamic.

For instance:

```
tsCopy codescores["Zain"] = "A+"; // ❌ string not assignable to number
```

We want a *typed* way to enforce both key and value consistency.

⚙️ 2️⃣ Introducing `Record<K, V>`

The `Record` utility type lets you describe an **object type with known key types and value types**.

```
tsCopy codetype Scores = Record<string, number>;
const marks: Scores = { Ali: 95, Fuzail: 90 };
```

✅ Each key must be a `string`.

✅ Each value must be a `number`.

✅ Fully type-safe, even for computed/dynamic keys.

Example — Typed Dictionary

```
tsCopy codetype CountryPopulation = Record<string, number>;

const population: CountryPopulation = {
  Pakistan: 240_000_000,
  India: 1_400_000_000,
};
```

If you try to assign a string to a numeric value:

```
tsCopy codepopulation["China"] = "large"; // ❌ Error
```

🧭 3️⃣ `Record` with Union Keys

When the keys are **known**, you can enforce exact key sets:

```
tsCopy codetype Direction = "N" | "S" | "E" | "W";
type Steps = Record<Direction, number>;

const moves: Steps = {
  N: 1,
  S: 2,
  E: 0,
  W: 1,
};
```

✅ TypeScript will warn you if you miss or mistype a key.

This pattern is excellent for constant lookups, such as grid moves, commands, or category mappings.

🧠 4️⃣ Index Signatures vs Record

Feature	Index Signature	`Record<K, V>`
Syntax	`{ [key: string]: V }`	`Record<K, V>`
Key type	usually `string` or `number`	explicit type parameter
Flexibility	Dynamic keys	Known/fixed keys allowed
Readability	Verbose	Cleaner and modern

Example equivalence:

```
tsCopy codetype Scores1 = { [key: string]: number };
type Scores2 = Record<string, number>; // same
```

🔎 5️⃣ Nested `Record`s

You can combine them for structured data:

```
tsCopy codetype StudentSubjects = Record<string, Record<string, number>
>;

const studentData: StudentSubjects = {
  Fuzail: { Math: 95, Physics: 88 },
  Ali: { Math: 85, English: 91 },
};
```

✅ Useful in hierarchical or grouped structures (common in problems like "Group Anagrams").

💡 6️⃣ Example — Word Frequency Counter

```
tsCopy codefunction wordCount(sentence: string): Record<string, number> {
  const words = sentence.split(" ");
  const freq: Record<string, number> = {};

  for (const word of words) {
    freq[word] = (freq[word] || 0) + 1;
  }

  return freq;
}
```

✅ Simple and readable.

✅ `Record` ensures all values are numbers, not random types.

🧮 7️⃣ Example — Group by Length

```
tsCopy codefunction groupByLength(words: string[]): Record<number, string
[]> {
  const groups: Record<number, string[]> = {};
```

```
  for (const word of words) {
    const len = word.length;
    groups[len] = groups[len] || [];
    groups[len].push(word);
  }

  return groups;
}
```

LeetCode-style use: grouping strings by property (e.g., *Group Anagrams*, *Word Pattern*).

🧠 8️⃣ `keyof` Operator — Extracting Keys of an Object Type

```
tsCopy codeinterface Student {
  name: string;
  age: number;
  grade: string;
}

type Keys = keyof Student; // "name" | "age" | "grade"
```

This allows TypeScript to reason about **what keys exist** on a type — crucial for writing flexible yet type-safe code.

Example — Safe Getter Function

```
tsCopy codefunction getValue<T, K extends keyof T>(obj: T, key: K): T[K] {
  return obj[key];
}

const student = { name: "Fuzail", age: 21 };
console.log(getValue(student, "name")); // ✅ string
```

```
console.log(getValue(student, "age")); // ✅ number
// console.log(getValue(student, "grade")); ❌ Error: not a key of student
```

✅ `T[K]` is an **indexed access type** → “value type at key `K` of type `T`”.

⚙️ 9️⃣ `typeof` — Creating Types from Objects

You can derive a type directly from an object:

```
tsCopy codeconst person = {
  name: "Ali",
  age: 22,
};

type PersonType = typeof person;
// { name: string; age: number; }
```

Combine with `keyof`:

```
tsCopy codetype PersonKeys = keyof typeof person; // "name" | "age"
```

This is extremely helpful for dynamic but type-safe access patterns.

🧩 🔟 Example — Dynamic Access and Updates

```
tsCopy codefunction updateValue<T, K extends keyof T>(obj: T, key: K, value:
T[K]) {
  obj[key] = value;
}

const user = { id: 1, active: true };
updateValue(user, "active", false); // ✅ works
```

✅ Ensures type correctness automatically.

✅ Prevents updating with incompatible types.

⚙️ 1️⃣1️⃣ Mapped Types — Transforming Objects

Mapped types let you **transform all keys or values** in a type.

Example — Make all properties optional:

```
tsCopy codetype Partial<T> = { [K in keyof T]?: T[K] };
```

Or make all properties readonly:

```
tsCopy codetype ReadonlyObj<T> = { readonly [K in keyof T]: T[K] };
```

Example usage:

```
tsCopy codeinterface Todo {
  title: string;
  done: boolean;
}

type OptionalTodo = Partial<Todo>;
type ReadonlyTodo = ReadonlyObj<Todo>;
```

🧠 1️⃣2️⃣ Advanced Example — Count Frequencies Using Keyof

```
tsCopy codefunction countByProperty<T, K extends keyof T>(
  items: T[],
  key: K
): Record<string, number> {
  const count: Record<string, number> = {};
  for (const item of items) {
    const value = String(item[key]);
    count[value] = (count[value] || 0) + 1;
  }
```

```
  return count;
}
```

✅ Can count by *any* property of the object dynamically.

✅ Perfect for flexible problem-solving patterns.

🧩 Exercises

Exercise 1

Define a type `GradeBook` that maps student names (`string`) to an object with:

- `score: number`
- `passed: boolean`

Then create a few student entries.

Exercise 2

Write a function:

```
tsCopy codefunction getKeys<T>(obj: T): (keyof T)[];
```

It should return an array of all keys in a type-safe way.

Exercise 3

Use a nested `Record` to store book categories and lists of book titles.

Example:

```
tsCopy code{
  Fiction: ["1984", "Brave New World"],
  Science: ["A Brief History of Time"]
}
```

Exercise 4

Write a generic `mergeObjects<T, U>(obj1: T, obj2: U): T & U`

that combines two objects safely.

Exercise 5

Create a function `filterByKey<T, K extends keyof T>(arr: T[], key: K, value: T[K]): T[]`

which filters an array of objects by a specific key-value pair.

🧭 Quick Summary

Concept	Use
`Record<K, V>`	Typed dictionary for dynamic or fixed keys
Index Signature	Custom shape for dynamic string/number keys
`keyof`	Extract all property names of a type
`typeof`	Create a type from an existing object
`T[K]`	Access value type by key
Mapped Types	Transform property types dynamically

🚀 Next Phase — **Session 6: Generics & Type Constraints**

Now that you know how to build and manipulate type-safe objects, the next step is **TypeScript Generics** — the real backbone of reusable, flexible LeetCode helpers.

We'll learn:

- What generics are and how they work under the hood.
- How to apply constraints (`extends`, `keyof`, `infer`).
- Write generic data structures like **Stack**, **Queue**, and **Graph Node**.
- Apply them directly to problem-solving patterns.

Would you like me to move forward to **Session 6: Generics & Type Constraints**,

or would you prefer I first include **visual memory tables** (like "Record vs keyof vs typeof comparison" and "generic access patterns") before continuing?

You said:

Ok go for session 6

ChatGPT said:

Perfect 👏 — you're moving steadily through the foundations.

Let's now step into **Session 6**, where TypeScript becomes really *powerful* for abstracting and scaling — this is the **Generics** session.

🧭 Session 6 — Generics in TypeScript

Generics let your functions, classes, and types **work with multiple data types** while maintaining **type safety**.

They're *parametric types* — meaning you write logic once and the compiler adapts it for whatever types you pass in.

🎯 Learning Goals

By the end of this session, you'll:

- Understand what generics are and why they exist.
- Learn to use generics in **functions**, **interfaces**, **classes**, and **constraints**.
- Apply generics to common **LeetCode-style patterns** (sorting, stacks, maps, etc.).
- Master the difference between **generic parameters** and **union types**.

🧩 1️⃣ What Are Generics?

Without generics, you lose flexibility or safety:

```
tsCopy codefunction identity(value: any): any {
  return value;
}

const result = identity(10);     // returns number, but type is `any`
```

This works but kills type inference — you could pass anything, and TypeScript won't warn you.

✅ Using **generics**:

```
tsCopy codefunction identity<T>(value: T): T {
  return value;
}

const result = identity<number>(10); // inferred as number
```

💡 The `<T>` is a **type variable** — it's a placeholder that stands for any type used during function call.

🧱 2️⃣ Generic Type Inference

You often don't need to explicitly specify `<T>` — TS can infer it automatically:

```
tsCopy codeconst result = identity("hello");  // TypeScript infers T = string
```

✅ In most LeetCode utilities, inference is automatic (cleaner code).

⚙️ 3️⃣ Generic Functions — Multiple Type Parameters

You can define **more than one type variable** if needed:

```
tsCopy codefunction merge<A, B>(objA: A, objB: B): A & B {
  return { ...objA, ...objB };
}

const user = merge({ name: "Fuzail" }, { age: 21 });
// type: { name: string; age: number }
```

💡 The returned type combines both (intersection).

🧩 4️⃣ Generic Constraints (`extends`)

Sometimes you don't want to allow *any* type, only those that meet specific conditions.

Example:

```
tsCopy codefunction getLength<T extends { length: number }>(arg: T): numb
er {
  return arg.length;
}

getLength("hello"); // ok
getLength([1, 2, 3]); // ok
getLength(123); // ❌ number has no 'length'
```

✅ Use `extends` to **limit** what kind of types `T` can be.

🧩 5️⃣ Generics in Interfaces

```
tsCopy codeinterface Box<T> {
  content: T;
}

const numberBox: Box<number> = { content: 42 };
const stringBox: Box<string> = { content: "Hi" };
```

💡 Common pattern in problem-solving:

```
tsCopy codeinterface KeyValue<K, V> {
  key: K;
  value: V;
}
```

🧩 6️⃣ Generics in Classes

Used frequently in **data structures** (Stack, Queue, LinkedList, etc.).

```
tsCopy codeclass Stack<T> {
  private items: T[] = [];

  push(item: T): void {
    this.items.push(item);
  }

  pop(): T | undefined {
    return this.items.pop();
  }
}

const numStack = new Stack<number>();
numStack.push(10);
```

```
numStack.push(20);
console.log(numStack.pop()); // 20
```

💡 In LeetCode, you might see similar patterns for **DFS, BFS, trees, graphs**, etc.

🧩 7️⃣ Generics with Default Types

If a type parameter isn't provided, you can assign a default.

```
tsCopy codeinterface ResponseData<T = any> {
  success: boolean;
  data: T;
}

const res1: ResponseData<string> = { success: true, data: "ok" };
const res2: ResponseData = { success: false, data: 404 }; // defaults to any
```

🧩 8️⃣ Generic Utility Functions — Real Use Cases

✅ Example 1: Find Minimum Element

```
tsCopy codefunction findMin<T extends number | string>(arr: T[]): T {
  return arr.reduce((min, val) => (val < min ? val : min));
}
```

✅ Example 2: Swap Values

```
tsCopy codefunction swap<A, B>(a: A, b: B): [B, A] {
  return [b, a];
}

const [x, y] = swap("apple", 10); // y: string, x: number
```

✅ Example 3: Filter by Property (Constraint)

```
tsCopy codefunction filterBy<T extends object, K extends keyof T>(
  arr: T[],
  key: K,
  value: T[K]
): T[] {
  return arr.filter((item) => item[key] === value);
}

const users = [
  { name: "Ali", age: 20 },
  { name: "Sara", age: 25 },
];

console.log(filterBy(users, "age", 25)); // [{ name: "Sara", age: 25 }]
```

🧠 9️⃣ Generics vs Union Types — Key Difference

Feature	Generics	Union Types
Reusable logic	✅ Yes	❌ No
Type consistency	Keeps same type across inputs/outputs	Can mix types
Example	`function id<T>(x:T):T`	`function id(x:number

🧩 1️⃣0️⃣ Advanced Example — Generic Map Wrapper

Let's wrap JS Map with strong types:

```
tsCopy codeclass TypedMap<K, V> {
  private map = new Map<K, V>();

  set(key: K, value: V): void {
    this.map.set(key, value);
```

```
  }

  get(key: K): V | undefined {
    return this.map.get(key);
  }

  entries(): [K, V][] {
    return Array.from(this.map.entries());
  }
}

const userMap = new TypedMap<number, string>();
userMap.set(1, "Fuzail");
userMap.set(2, "Ayesha");
```

💡 This mirrors many **LeetCode Map-based problems**, but with full type safety.

🧩 Exercises

Exercise 1

Create a generic function `reverseArray<T>(arr: T[]): T[]` that reverses any array.

Exercise 2

Create a generic interface `Pair<A, B>` and write a function `swapPair(pair: Pair<A, B>): Pair<B, A>`.

Exercise 3

Implement a `Queue<T>` class with:

- `enqueue(item: T): void`
- `dequeue(): T | undefined`
- `size(): number`

Exercise 4

Build a function `pluck<T, K extends keyof T>(arr: T[], key: K): T[K][]`

→ Extracts property `key` from every object in an array.

Exercise 5

Write a `mergeObjects<A, B>(a: A, b: B): A & B` function that merges two objects into one.

🏁 Summary

Concept	Description
`<T>`	Declares a generic type parameter
`extends`	Restricts what types can be passed
`keyof`	Used to reference property names dynamically
Inference	TypeScript deduces types automatically
Default generics	Set fallback types
Use cases	Data structures, helpers, reusable utilities

✅ **Next Session (7)** → *Utility Types* — You'll learn how TypeScript provides **built-in type transformers** like `Partial`, `Pick`, `Omit`, `Readonly`, and how they help in shaping data for algorithms cleanly.

Would you like me to **move on to Session 7: Utility Types**?

...

You said:

Yes go further to session 7

ChatGPT said:

Perfect 🚀

You've now mastered **Generics**, which is one of the most powerful features of TypeScript.

Now let's take that foundation and make it *practical* with **TypeScript Utility Types** — tools that allow you to transform and manipulate existing types elegantly.

🧭 Session 7 — Utility Types (Built-in Type Transformers)

Utility types are **predefined generic helpers** that simplify type manipulation.

They're essential in both **real-world development** and **LeetCode problem-solving**, where you often need to reshape objects or create partial, readonly, or selective variants.

🎯 Learning Goals

By the end of this session, you'll:

- Understand what utility types are and how they relate to generics.
- Learn how to use key built-ins like `Partial`, `Required`, `Readonly`, `Pick`, `Omit`, `Record`, and `ReturnType`.
- Learn when and *why* to use each type in problem-solving or modeling data.

🧩 1️⃣ What Are Utility Types?

Utility types are **built on top of generics** — they're reusable functions for transforming existing types.

Example:

```
tsCopy codeinterface User {
  id: number;
  name: string;
  email?: string;
}
```

You can create variations without rewriting interfaces.

🧱 2️⃣ `Partial<T>`

Makes all properties **optional**.

```
tsCopy codetype PartialUser = Partial<User>;

// Equivalent to:
// { id?: number; name?: string; email?: string; }
```

Useful when updating part of an object:

```
tsCopy codefunction updateUser(id: number, update: Partial<User>) {
  // update may contain one or more fields
}
updateUser(1, { name: "Fuzail" });
```

💡 **LeetCode use case:** When you're gradually building an object, such as a frequency map or configuration object.

🧱 3️⃣ `Required<T>`

Opposite of `Partial`.

Makes all properties **mandatory**.

```
tsCopy codetype CompleteUser = Required<User>;
// { id: number; name: string; email: string; }
```

💡 Ensures your final structure is fully complete before processing.

🧱 4️⃣ `Readonly<T>`

Prevents mutation of object properties.

```
tsCopy codeconst config: Readonly<User> = { id: 1, name: "Ali", email: "x@y.c
om" };
config.name = "Zara"; // ❌ Error
```

💡 Ideal when working with **constant configurations** or **graph nodes** that shouldn't mutate after initialization.

🧱 5️⃣ `Pick<T, K extends keyof T>`

Selects specific properties from a type.

```
tsCopy codetype BasicInfo = Pick<User, "id" | "name">;
// { id: number; name: string; }
```

💡 **When to use:** You only need a small subset (like ID and name) — this keeps types lightweight.

🧱 6️⃣ `Omit<T, K extends keyof T>`

Removes certain properties from a type.

```
tsCopy codetype WithoutEmail = Omit<User, "email">;
// { id: number; name: string; }
```

💡 Great when a problem doesn't require all data fields (e.g., trimming unnecessary info).

🧱 7️⃣ `Record<K, T>`

You've already learned this earlier — it creates an **object type with specific key-value pairs**.

```
tsCopy codetype ScoreCard = Record<string, number>;
const scores: ScoreCard = { Fuzail: 95, Sara: 88 };
```

💡 Most useful for **frequency maps, adjacency lists, or memoization**.

🧱 8️⃣ `Exclude<T, U>`

Removes specific types from a union.

```
tsCopy codetype Status = "pending" | "success" | "error";
type NonError = Exclude<Status, "error">;
// "pending" | "success"
```

💡 Simplifies conditional logic or narrowing down results in algorithms.

🧱 9️⃣ `Extract<T, U>`

Opposite of `Exclude` — *keeps only* matching types.

```
tsCopy codetype Status = "pending" | "success" | "error";
type GoodStatus = Extract<Status, "success" | "pending">;
// "pending" | "success"
```

🧱 🔟 `ReturnType<T>`

Infers the **return type** of a function.

```
tsCopy codefunction makeUser() {
  return { id: 1, name: "Fuzail" };
}

type UserReturn = ReturnType<typeof makeUser>;
// { id: number; name: string }
```

💡 Handy when you want to type a variable that depends on a function's output.

🧩 1️⃣1️⃣ `Parameters<T>`

Extracts the **parameter types** of a function.

```
tsCopy codefunction greet(name: string, age: number) { }
type GreetParams = Parameters<typeof greet>;
// [string, number]
```

🧩 1️⃣2️⃣ `ReadonlyArray<T>`

Immutable version of arrays.

```
tsCopy codeconst arr: ReadonlyArray<number> = [1, 2, 3];
arr.push(4); // ❌ Error
```

💡 Helps in LeetCode problems where arrays must not be modified in-place.

🧠 1️⃣3️⃣ Combining Utility Types

You can combine multiple utility types for advanced transformations.

```
tsCopy codetype SafeUser = Readonly<Partial<User>>;
// { id?: number; name?: string; email?: string } (all readonly)
```

Or even:

```ts
tsCopy codetype OptionalNoEmail = Omit<Partial<User>, "email">;
```

🧩 1️⃣4️⃣ Real Problem-Solving Examples

✅ Example 1: Update Partial Object

```ts
tsCopy codefunction applyPatch<T>(obj: T, patch: Partial<T>): T {
  return { ...obj, ...patch };
}

const user = { name: "Ali", age: 20 };
const updated = applyPatch(user, { age: 21 });
```

✅ Example 2: Safe Immutable Configuration

```ts
tsCopy codetype Config = Readonly<{ port: number; mode: "dev" | "prod" }>;
const serverConfig: Config = { port: 8080, mode: "dev" };
// serverConfig.port = 3000 ❌
```

✅ Example 3: Extract Function Info

```ts
tsCopy codefunction calculate(a: number, b: number) {
  return a + b;
}
type Args = Parameters<typeof calculate>; // [number, number]
type Result = ReturnType<typeof calculate>; // number
```

🧩 Exercises

Exercise 1

Create an interface `Student` with `name`, `age`, and `grade`.

Then:

1. Create a `Partial<Student>` called `updateStudent`.
2. Create a `Readonly<Student>` called `immutableStudent`.
3. Create an `Omit<Student, "grade">` called `basicInfo`.

Exercise 2

Given:

```
tsCopy codefunction makePoint(x: number, y: number) {
  return { x, y };
}
```

Use utility types to extract both:

- Function **parameter types**.
- Function **return type**.

Exercise 3

Create a function `mergeObjects<A, B>(a: A, b: B): A & B` again,

but this time ensure its result is **Readonly** using `Readonly<ReturnType<typeof mergeObjects>>`.

Exercise 4

Make a `Partial` + `Readonly` version of this type:

```
tsCopy codeinterface Config {
  host: string;
  port: number;
  secure: boolean;
}
```

Call it `SafeConfig`.

🏁 Summary

Utility	Purpose
`Partial<T>`	Make all fields optional
`Required<T>`	Make all fields mandatory
`Readonly<T>`	Prevent modification
`Pick<T,K>`	Select fields
`Omit<T,K>`	Exclude fields
`Record<K,T>`	Create dictionary-like types
`Exclude<T,U>`	Remove members from union
`Extract<T,U>`	Keep only certain members
`ReturnType<T>`	Get function’s return type
`Parameters<T>`	Get function parameter types
`ReadonlyArray<T>`	Immutable array

🧩 Next Step

Next up is **Session 8 → Advanced Type Operators and Conditional Types**,

where we’ll unlock the inner mechanics behind `keyof`, `typeof`, `in`, conditional types, and type inference patterns.

This is where we’ll start learning how to build **your own utility types**, like:

- `Mutable<T>`
- `NonNullable<T>`
- `DeepPartial<T>`
- and more.

Would you like me to move on to **Session 8 (Advanced Type Operators & Conditional Types)**?

You said:

Yes move further to session 8

ChatGPT said:

Awesome 💪 — you're doing great so far!

Now that you've learned *how to use* built-in utility types, it's time to learn *how they actually work*.

This session dives deep into **TypeScript's Advanced Type Operators & Conditional Types** — the *core logic* that powers TypeScript's type system.

🧭 Session 8 — Advanced Type Operators & Conditional Types

This is one of the most advanced (and rewarding) TypeScript lessons — mastering it will help you:

- Read and understand any complex TS code.
- Build your own custom utility types.

- Sharpen type inference for data structures and problem-solving.

🎯 Learning Goals

By the end of this session, you'll:

- Master `keyof`, `typeof`, and `in` operators.
- Understand **indexed access types** and **mapped types**.
- Learn **conditional types** and **infer** keyword.
- Build **custom utility types** (like `DeepPartial`, `Mutable`, etc.).

🧩 1️⃣ `keyof` Operator — Extract Keys of a Type

`keyof` returns a union of property names from a type.

```
tsCopy codeinterface User {
  id: number;
  name: string;
  active: boolean;
}

type UserKeys = keyof User; // "id" | "name" | "active"
```

✅ Used in generic utilities:

```
tsCopy codefunction getValue<T, K extends keyof T>(obj: T, key: K): T[K] {
  return obj[key];
}

const user = { id: 1, name: "Fuzail", active: true };
const nameValue = getValue(user, "name"); // string
```

💡 *LeetCode analogy:* If you have an object representing graph nodes or frequency maps, this helps safely access known keys.

🧩 2️⃣ `typeof` Operator — Get Type from a Value

At runtime, `typeof` gives *data type*.

At **compile time** (in TypeScript), it extracts the **type** of a variable or constant.

```
tsCopy codeconst config = { port: 8080, secure: true };
type ConfigType = typeof config; // { port: number; secure: boolean }
```

💡 Use this when you want your types to mirror existing variables.

🧩 3️⃣ `in` Operator — Used in Mapped Types

The `in` keyword iterates over keys in a union type.

```
tsCopy codetype Directions = "N" | "S" | "E" | "W";

type MoveCount = {
  [D in Directions]: number;
};

// { N: number; S: number; E: number; W: number; }
```

💡 *This is how utility types like `Record<K,V>` are actually built.*

🧩 4️⃣ Indexed Access Types (`T[K]`)

Lets you reference a specific property type inside another type.

```
tsCopy codeinterface User {
  id: number;
  name: string;
  active: boolean;
}
```

```
type IdType = User["id"]; // number
type NameType = User["name"]; // string
```

You can even combine it with `keyof`:

```
tsCopy codetype ValueTypes = User[keyof User]; // number | string | boolean
```

💡 Great for extracting property value unions dynamically.

🧩 5️⃣ Mapped Types

Mapped types transform all properties of an existing type.

Example — make everything optional manually:

```
tsCopy codetype MyPartial<T> = {
  [P in keyof T]?: T[P];
};
```

Equivalent to built-in `Partial<T>`!

💡 You can use any transformation logic here — like adding modifiers, converting types, etc.

🧩 6️⃣ Conditional Types

Syntax:

```
tsCopy codeT extends U ? X : Y
```

Think of it as an **if-else** for types.

Example:

```
tsCopy codetype IsNumber<T> = T extends number ? "yes" : "no";
type A = IsNumber<number>; // "yes"
type B = IsNumber<string>; // "no"
```

💡 *Conditional types enable branching logic inside type definitions.*

🧩 7️⃣ Conditional Types with Inference (`infer`)

`infer` lets TypeScript **capture** a type from another type.

Example:

```
tsCopy codetype ReturnTypeOf<T> = T extends (...args: any[]) => infer R ? R : never;

function greet() {
  return "hello";
}
type GreetReturn = ReturnTypeOf<typeof greet>; // string
```

💡 That's literally how built-in `ReturnType<T>` works under the hood.

🧩 8️⃣ Distributive Conditional Types

Conditional types distribute over unions automatically.

```
tsCopy codetype ToArray<T> = T extends any ? T[] : never;

type Result = ToArray<number | string>;
// (number | string)[] — distributes into number[] | string[]
```

💡 Useful when transforming every member of a union separately.

🧩 9️⃣ Custom Utility Types (Building Your Own)

Now let's make some of the built-in ones yourself 👇

✅ `MyReadonly<T>`

```
tsCopy codetype MyReadonly<T> = {
  readonly [P in keyof T]: T[P];
```

```
};
```

✅ `MyRequired<T>`

```
tsCopy codetype MyRequired<T> = {
  [P in keyof T]-?: T[P];
};
```

✅ `MyPick<T, K extends keyof T>`

```
tsCopy codetype MyPick<T, K extends keyof T> = {
  [P in K]: T[P];
};
```

✅ `MyOmit<T, K extends keyof any>`

```
tsCopy codetype MyOmit<T, K extends keyof any> = {
  [P in keyof T as P extends K ? never : P]: T[P];
};
```

✅ `Mutable<T>` — **opposite of** `Readonly<T>`

```
tsCopy codetype Mutable<T> = {
  -readonly [P in keyof T]: T[P];
};
```

✅ `NonNullable<T>`

```
tsCopy codetype NonNullable<T> = T extends null | undefined ? never : T;
```

🧩 1️⃣0️⃣ Deep Utility Types (Advanced Challenge)

✅ `DeepPartial<T>`

Makes all properties (and nested ones) optional.

```
tsCopy codetype DeepPartial<T> = {
  [P in keyof T]?: T[P] extends object ? DeepPartial<T[P]> : T[P];
};

interface Profile {
  name: string;
  address: {
    city: string;
    zip: number;
  };
}

const update: DeepPartial<Profile> = {
  address: { city: "Lahore" },
};
```

💡 This pattern is powerful for recursive object transformations — common in configuration handling or dynamic updates.

🧩 1️⃣1️⃣ Example — Using Conditional Types in Practice

Use case: filter numbers from an array type.

```
tsCopy codetype FilterNumbers<T> = T extends number ? T : never;
type OnlyNumbers = FilterNumbers<string | number | boolean>;
// number
```

You can generalize this:

```
tsCopy codetype ExcludeType<T, U> = T extends U ? never : T;
type WithoutBoolean = ExcludeType<string | number | boolean, boolean>;
// string | number
```

🧩 1️⃣2️⃣ Real LeetCode Use Case — Type-Safe Stack

Combining multiple advanced tools:

```
tsCopy codeclass Stack<T> {
  private items: T[] = [];

  push<U extends T>(item: U) {
    this.items.push(item);
  }

  pop(): T | undefined {
    return this.items.pop();
  }
}
```

✅ Enforces that items added match the stack's generic constraint

✅ Uses generic inheritance `<U extends T>` for flexibility

🧠 Exercises

Exercise 1

Define a type `GetKeys<T>` that returns all keys of `T` as a union.

Exercise 2

Implement a type `OptionalExcept<T, K extends keyof T>`

→ Makes all properties optional *except* the ones in `K`.

Exercise 3

Create a type `FunctionReturn<T>` that extracts a function's return type manually using `infer`.

Exercise 4

Build your own version of `DeepPartial<T>` and test it on a nested object.

Exercise 5

Implement a `Mutable<T>` type and use it on a `Readonly` object.

🏁 Summary

Operator	Description	Example
`keyof`	Extracts property names	`keyof User → "id"
`typeof`	Extracts type of variable	`typeof config → {port:number}`
`in`	Iterate keys in mapped types	`[K in keyof T]`
`T[K]`	Access value type	`User["name"] → string`
`extends`	Conditional type check	`T extends U ? X : Y`
`infer`	Capture inferred type	`infer R` inside condition
Mapped types	Transform all keys	`[P in keyof T]?: T[P]`

🚀 Next Step

You've now reached an **expert level** in TypeScript's type system.

The next session — **Session 9: TypeScript Patterns for LeetCode Problems** — will show **how to apply all this theory** to actual **algorithmic problem solving**, such as:

- Hash maps, stacks, and queues
- Trees and graphs
- Priority queues & generics
- Memoization helpers
- Type-safe recursion

Would you like me to move ahead to **Session 9 — LeetCode Problem-Solving Patterns in TypeScript**?

You said:

Yes move further I am excited

ChatGPT said:

Edited Type Script Modern Types — Lesson Series For Leet Code Problem Solving

Session 9 on **Utility Types** is now complete — it includes detailed explanations, code samples, and practical exercises.

Would you like me to move on to **Session 10: Applying Modern Types in LeetCode Problems**, where we'll use all of these modern TypeScript types to solve actual coding challenges (like *Two Sum*, *Group Anagrams*, and *LRU Cache*)?

You said:

Yes move to session 10

ChatGPT said:

Edited Type Script Modern Types — Lesson Series For Leet Code Problem Solving

Session 10 on **Applying Modern Types in LeetCode Problems** is complete — it includes five real-world problems, detailed explanations, and type-focused exercises.

Would you like to move on to **Session 11: TypeScript Patterns for Algorithms & Data Structures** next?

You said:

yes move to session 11

ChatGPT said:

Type Script Modern Types — Lesson Series For Leet Code Problem Solving

Copy

Edit

Download

Session 11 — TypeScript Patterns for Algorithms & Data Structures

Now that you've seen how to use modern types in real LeetCode problems, we'll go deeper into **common algorithmic patterns** — like Sliding Window, Two Pointers, Recursion, DFS/BFS, and Hashing — and how to model them effectively in **TypeScript**.

Each section includes type-safe templates, practical examples, and exercises.

🧩 1. Sliding Window Pattern

Used for problems involving **contiguous subarrays or substrings** (like finding max sum or longest substring without repetition).

Key Type Concepts Used:

- `Record<string, number>` for character frequency.
- Tuples for window indices.

```
function lengthOfLongestSubstring(s: string): number {
const seen: Record<string, number> = {};
let left = 0;
let maxLen = 0;
for (let right = 0; right < s.length; right++) {
const char = s[right];
if (seen[char] !== undefined && seen[char] >= left) {
left = seen[char] + 1;
}
seen[char] = right;
maxLen = Math.max(maxLen, right - left + 1);
}
return maxLen;
}
```

🧠 **Why modern typing helps:**

- The `Record<string, number>` type cleanly models last-seen positions.
- Tuple `[left, right]` conceptually defines the window bounds.

✅ **Exercise:**
Implement a function `maxSumSubarray(nums: number[], k: number): number` using the sliding window technique.

🧩 2. Two Pointer Pattern

Used for problems involving **sorted arrays or linked lists** (like merging, removing duplicates, or finding pairs).

Example — Remove Duplicates from Sorted Array

```
function removeDuplicates(nums: number[]): number {
let left = 0;
```

```
for (let right = 1; right < nums.length; right++) {
if (nums[left] !== nums[right]) {
left++;
nums[left] = nums[right];
}
}
return left + 1;
}
```

🧠 **Typing Tip:**
Even though no explicit TypeScript features are used, static typing ensures both pointers always operate on valid indices.

✅ **Exercise:**
Use two pointers to implement `isPalindrome(s: string): boolean` while ignoring non-alphanumeric characters.

🧩 3. Depth-First Search (DFS)

Used in graph/tree traversal problems.

Key Type Concepts Used:

- Recursive function typing.
- `Map` and `Set` for visited nodes.

```
function dfs(graph: Map<number, number[]>, start: number, visited = new Set<number>()): void {
if (visited.has(start)) return;
visited.add(start);
console.log(start);
for (const neighbor of graph.get(start) ?? []) {
dfs(graph, neighbor, visited);
}
```

```
}
const graph = new Map<number, number[]>([
[1, [2, 3]],
[2, [4]],
[3, []],
[4, []],
]);
dfs(graph, 1);
```

🧠 **Why modern typing helps:**

- `Map<number, number[]>` enforces that each key maps to an array of numbers.
- Recursive type safety prevents invalid calls.

✅ **Exercise:**
Implement a recursive function `sumTree(root: TreeNode | null): number` where `TreeNode` is an interface:

```
interface TreeNode {
val: number;
left: TreeNode | null;
right: TreeNode | null;
}
```

🧩 4. Breadth-First Search (BFS)

Used for **shortest path or level order traversal** problems.

Example — Level Order Traversal

```
function levelOrder(root: TreeNode | null): number[][] {
if (!root) return [];
const queue: TreeNode[] = [root];
const result: number[][] = [];
```

```
while (queue.length > 0) {
const level: number[] = [];
const size = queue.length;
for (let i = 0; i < size; i++) {
const node = queue.shift()!;
level.push(node.val);
if (node.left) queue.push(node.left);
if (node.right) queue.push(node.right);
}
result.push(level);
}
return result;
}
```

🧠 **Why modern typing helps:**

- The queue has a strict `TreeNode[]` type, ensuring consistent element types.

✅ **Exercise:**
Convert this BFS to return a flattened array instead of a 2D array.

🧩 5. Hashing Pattern

Used for **frequency-based** or **lookup-based** problems.

Example — Find Duplicates

```
function containsDuplicate(nums: number[]): boolean {
const seen = new Set<number>();
for (const n of nums) {
if (seen.has(n)) return true;
seen.add(n);
}
```

```
return false;
}
```

✅ **Exercise:**
Implement a function `firstUniqueChar(s: string): number` that returns the first index of a non-repeating character using a frequency `Map`.

🧩 6. Dynamic Programming (DP) Pattern

Used for optimization problems (like Fibonacci, climbing stairs, etc.)

Example — Climbing Stairs

```
function climbStairs(n: number): number {
const dp: number[] = new Array(n + 1).fill(0);
dp[0] = 1;
dp[1] = 1;
for (let i = 2; i <= n; i++) {
dp[i] = dp[i - 1] + dp[i - 2];
}
return dp[n];
}
```

🧠 **Typing advantage:**

- Using `number[]` enforces consistent numeric operations.
- Helpful for memoization with predictable return types.

✅ **Exercise:**
Implement `fib(n: number): number` recursively with memoization using `Record<number, number>`.

🧠 Session Summary

By now, you've learned how to apply TypeScript typing across key algorithmic patterns:

Pattern	Key Types	Use Case
Sliding Window	Record, Tuples	Substrings / Subarrays
Two Pointers	Arrays	Sorted data / pairs
DFS	Map, Set, Recursion	Graphs / Trees
BFS	Queues, Arrays	Shortest path, levels
Hashing	Map, Set	Lookups / Frequency
DP	Arrays, Records	Optimization problems

Would you like to continue to **Session 12: Advanced Data Structures with TypeScript (Linked Lists, Trees, Graphs)** next?

Session 11 on **TypeScript Patterns for Algorithms & Data Structures** is now complete — it includes in-depth examples, type reasoning, and targeted exercises for major algorithmic patterns.

Would you like to continue to **Session 12: Advanced Data Structures with TypeScript (Linked Lists, Trees, Graphs)** next?

...

You said:

Yes go for session 12

ChatGPT said:

Perfect ⚡ Let's dive into **Session 12 — Advanced Data Structures with TypeScript (Linked Lists, Trees, Graphs)**.

This is where you learn to **build custom data structures** (like in LeetCode "Design" problems) while keeping everything **strongly typed** using interfaces, classes, and generics.

🧭 Session 12 Overview

You'll learn:

1. How to define **Linked Lists** using generics
2. How to model **Binary Trees** and traverse them
3. How to implement **Graphs** using `Map` and `Set`
4. When to use **interfaces vs classes**
5. How to make your data structures reusable and type-safe

🔷 1. Linked List

A **Linked List** is a chain of nodes, where each node points to the next.

Interface + Class

```
tsCopy codeinterface ListNode<T> {
  value: T;
  next: ListNode<T> | null;
}

class LinkedList<T> {
  head: ListNode<T> | null = null;

  append(value: T): void {
    const newNode: ListNode<T> = { value, next: null };

    if (!this.head) {
      this.head = newNode;
```

```
      return;
    }

    let current = this.head;
    while (current.next) {
      current = current.next;
    }
    current.next = newNode;
  }

  print(): void {
    let current = this.head;
    const result: T[] = [];
    while (current) {
      result.push(current.value);
      current = current.next;
    }
    console.log(result.join(" → "));
  }
}

// Usage:
const list = new LinkedList<number>();
list.append(10);
list.append(20);
list.append(30);
list.print(); // 10 → 20 → 30
```

🧠 **Typing benefits:**

- `T` makes the list generic (you can have `LinkedList<string>` or `LinkedList<number>`).
- Prevents inserting mismatched data types.

✅ **Exercise:**

Add a method `find(value: T): boolean` that returns true if a value exists in the list.

🔷 2. Binary Tree

A **Binary Tree** consists of nodes with up to two children: left and right.

```
tsCopy codeinterface TreeNode<T> {
  val: T;
  left: TreeNode<T> | null;
  right: TreeNode<T> | null;
}

class BinaryTree<T> {
  root: TreeNode<T> | null = null;

  insert(val: T): void {
    const newNode: TreeNode<T> = { val, left: null, right: null };
    if (!this.root) {
      this.root = newNode;
      return;
    }

    const queue: TreeNode<T>[] = [this.root];
    while (queue.length) {
      const node = queue.shift()!;
      if (!node.left) {
        node.left = newNode;
        return;
      } else if (!node.right) {
        node.right = newNode;
        return;
      } else {
        queue.push(node.left, node.right);
      }
    }
  }

  inorder(node = this.root): void {
```

```
    if (!node) return;
    this.inorder(node.left);
    console.log(node.val);
    this.inorder(node.right);
  }
}

// Usage:
const tree = new BinaryTree<number>();
tree.insert(1);
tree.insert(2);
tree.insert(3);
tree.inorder(); // 2 1 3
```

🧠 **Typing benefits:**

- `TreeNode<T>` keeps the tree fully generic.
- Recursion type safety ensures only valid node structures are traversed.

✅ **Exercise:**

Add methods for **preorder** and **postorder** traversals using recursion.

🔷 3. Graph (Adjacency List Representation)

Graphs are perfect for representing networks, relationships, or routes.

Generic Graph Implementation

```
tsCopy codeclass Graph<T> {
  private adjacencyList: Map<T, Set<T>> = new Map();

  addVertex(vertex: T): void {
    if (!this.adjacencyList.has(vertex)) {
      this.adjacencyList.set(vertex, new Set());
    }
  }
```

```
  addEdge(v1: T, v2: T): void {
    this.addVertex(v1);
    this.addVertex(v2);
    this.adjacencyList.get(v1)!.add(v2);
    this.adjacencyList.get(v2)!.add(v1); // undirected
  }

  bfs(start: T): void {
    const visited = new Set<T>();
    const queue: T[] = [start];

    while (queue.length) {
      const vertex = queue.shift()!;
      if (!visited.has(vertex)) {
        console.log(vertex);
        visited.add(vertex);
        for (const neighbor of this.adjacencyList.get(vertex) || []) {
          queue.push(neighbor);
        }
      }
    }
  }
}

// Usage:
const g = new Graph<string>();
g.addEdge("A", "B");
g.addEdge("A", "C");
g.addEdge("B", "D");
g.bfs("A"); // A B C D
```

🧠 **Typing benefits:**

- `Map<T, Set<T>>` makes it fully generic for any node type.
- No need for manual key coercion — keys can be objects or strings.

✅ **Exercise:**

Add a `dfs(start: T)` method using recursion.

🔷 4. When to Use Interfaces vs Classes

Use Case	Prefer **Interface**	Prefer **Class**
Structural typing	✅	❌
Creating blueprints only	✅	❌
Implementation logic or methods	❌	✅
Inheritance (extend)	✅	✅
Real data structure instances	❌	✅

Example:

- `interface TreeNode<T>` → defines structure.
- `class BinaryTree<T>` → defines implementation.

🔷 5. Bonus: Generic Stack & Queue

```
tsCopy codeclass Stack<T> {
  private items: T[] = [];

  push(item: T) { this.items.push(item); }
  pop(): T | undefined { return this.items.pop(); }
  peek(): T | undefined { return this.items[this.items.length - 1]; }
  isEmpty(): boolean { return this.items.length === 0; }
}

class Queue<T> {
  private items: T[] = [];
```

```
  enqueue(item: T) { this.items.push(item); }
  dequeue(): T | undefined { return this.items.shift(); }
  isEmpty(): boolean { return this.items.length === 0; }
}
```

🧠 Great practice for type-safe stack/queue operations in BFS/DFS or simulation problems.

✅ **Exercise:**

Implement a **Queue using two stacks** (`Stack<T>` instances).

🧠 Session Summary

You've now mastered:

- ✅ Defining reusable **generic data structures**
- ✅ Differentiating between **interfaces and classes**
- ✅ Applying **Maps, Sets, and Generics** in structural design

You are now ready for **Session 13 → Advanced TypeScript Patterns in Problem Solving** — where we combine these data structures and TypeScript features into more complex problems like **LRU Cache**, **Graph Traversals**, and **Dynamic Programming with Memoization**.

Would you like to continue to **Session 13: Advanced TypeScript Problem Solving Patterns** next?

You said:

yes move

ChatGPT said:

🔥 Awesome — let's dive into **Session 13: Advanced TypeScript Problem Solving Patterns**.

This session focuses on how to **apply modern TypeScript features** (like generics, utility types, discriminated unions, and classes) to **real-world algorithmic challenges** — the kind you'll encounter on **LeetCode and coding interviews**.

🧭 Session 13 Overview

You'll learn how to:

1. Combine advanced types with algorithmic thinking
2. Use **TypeScript's type system** for correctness and readability
3. Apply **memoization, caching, and object typing** in DP and graph problems
4. Model **real-world LeetCode problems** with fully typed solutions

---

🔷 1. LRU Cache (LeetCode #146)

The **LRU Cache** is a classic design problem testing data structures + logic.

We'll use `Map` to maintain insertion order (since ES6 `Map` preserves it).

Fully Typed Implementation

```
tsCopy codeclass LRUCache<K, V> {
  private capacity: number;
```

```
  private cache: Map<K, V>;

  constructor(capacity: number) {
    this.capacity = capacity;
    this.cache = new Map();
  }

  get(key: K): V | -1 {
    if (!this.cache.has(key)) return -1;

    // Move accessed item to end (most recently used)
    const value = this.cache.get(key)!;
    this.cache.delete(key);
    this.cache.set(key, value);

    return value;
  }

  put(key: K, value: V): void {
    if (this.cache.has(key)) this.cache.delete(key);

    // Evict oldest if capacity exceeded
    if (this.cache.size === this.capacity) {
      const oldestKey = this.cache.keys().next().value;
      this.cache.delete(oldestKey);
    }

    this.cache.set(key, value);
  }

  print() {
    console.log([...this.cache.entries()]);
  }
}

// Usage:
```

```
const lru = new LRUCache<string, number>(3);
lru.put("a", 1);
lru.put("b", 2);
lru.put("c", 3);
lru.get("a"); // refresh "a"
lru.put("d", 4); // removes "b"
lru.print(); // [["c",3],["a",1],["d",4]]
```

🧠 **Typing Insight:**

- `K` and `V` make it reusable for any key/value types.
- Strict key typing avoids wrong cache keys.

✅ **Exercise:**

Add a method `has(key: K): boolean` that checks for existence without reordering items.

🔷 2. Dynamic Programming with Memoization

Let's type a **Top-Down Fibonacci** using memoization.

```
tsCopy codefunction fib(n: number, memo: Record<number, number> = {}): number {
  if (n <= 1) return n;
  if (n in memo) return memo[n];
  memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
  return memo[n];
}
```

🧠 **Typing details:**

- `Record<number, number>` → explicitly defines a dictionary-like object with numeric keys.
- TS catches if you accidentally store non-number results.

✅ **Exercise:**

Implement a **memoized factorial function** using the same pattern.

🔷 3. Graph Traversal with Types

We'll extend our graph from before and write a **typed DFS traversal**.

```
tsCopy codeclass Graph<T> {
  private adj: Map<T, Set<T>> = new Map();

  addEdge(u: T, v: T): void {
    if (!this.adj.has(u)) this.adj.set(u, new Set());
    if (!this.adj.has(v)) this.adj.set(v, new Set());
    this.adj.get(u)!.add(v);
    this.adj.get(v)!.add(u);
  }

  dfs(start: T): void {
    const visited = new Set<T>();
    const explore = (node: T) => {
      if (visited.has(node)) return;
      console.log(node);
      visited.add(node);
      for (const neighbor of this.adj.get(node) || []) explore(neighbor);
    };
    explore(start);
  }
}

// Usage:
const g = new Graph<number>();
g.addEdge(1, 2);
g.addEdge(2, 3);
g.addEdge(1, 4);
g.dfs(1);
```

🧠 **Typing highlights:**

- `Set<T>` and `Map<T, Set<T>>` ensure strong consistency.

- Prevents invalid graph edges due to type mismatches.

✅ **Exercise:**

Modify this graph to make it **directed** and add a method `getNeighbors(node: T): T[]`.

🔷 4. Type Guards and Discriminated Unions in Problem Solving

You can use **union types** to simplify problems with multiple possible shapes of data.

Example: You're designing a function that handles **shapes** for geometry problems.

```
tsCopy codetype Shape =
  | { kind: "circle"; radius: number }
  | { kind: "square"; side: number }
  | { kind: "rectangle"; width: number; height: number };

function getArea(shape: Shape): number {
  switch (shape.kind) {
    case "circle": return Math.PI * shape.radius ** 2;
    case "square": return shape.side ** 2;
    case "rectangle": return shape.width * shape.height;
    default: const _exhaustiveCheck: never = shape;
             return _exhaustiveCheck;
  }
}
```

🧠 **Why this matters for algorithms:**

- Discriminated unions are perfect for **decision-based problems**, e.g., handling multiple input types.
- The `never` type ensures **compile-time exhaustiveness checking**.

✅ **Exercise:**

Add a new shape `"triangle"` and ensure TypeScript forces you to update all cases.

🔷 5. Using Tuples in Graph + DP Problems

Tuples are great for **pair-based logic** — like coordinate systems, ranges, or DP states.

Example: **LeetCode #994 – Rotting Oranges**

```
tsCopy codefunction orangesRotting(grid: number[][]): number {
  const queue: [number, number, number][] = []; // [row, col, minutes]
  const rows = grid.length;
  const cols = grid[0].length;

  for (let r = 0; r < rows; r++) {
    for (let c = 0; c < cols; c++) {
      if (grid[r][c] === 2) queue.push([r, c, 0]);
    }
  }

  const directions = [[1,0], [-1,0], [0,1], [0,-1]];
  let minutes = 0;

  while (queue.length) {
    const [r, c, time] = queue.shift()!;
    minutes = Math.max(minutes, time);

    for (const [dr, dc] of directions) {
      const nr = r + dr, nc = c + dc;
      if (grid[nr]?.[nc] === 1) {
        grid[nr][nc] = 2;
        queue.push([nr, nc, time + 1]);
      }
    }
  }

  return grid.some(row => row.includes(1)) ? -1 : minutes;
}
```

🧠 **Typing benefit:**

- `[number, number, number]` tuple explicitly defines the structure of your queue.
- You can't accidentally enqueue a wrong data shape.

✅ **Exercise:**

Refactor this solution to use a helper function with a strongly typed return value.

🧠 Session Summary

You've now learned:

- ✅ How to use **TypeScript's type system** to strengthen algorithmic correctness
- ✅ How to model **LRU caches, DP tables, graphs**, and **discriminated unions**
- ✅ How to write **real-world LeetCode-level solutions** with strong typing

---

Next up is **Session 14: TypeScript Patterns for Dynamic Programming & Recursion**,

where we'll focus deeply on **recursive patterns**, **memoization**, and **state typing** (for problems like `Climbing Stairs`, `House Robber`, and `Knapsack`).

Would you like me to continue to **Session 14**?

...

You said:

yes move next

ChatGPT said:

Perfect 💪 — Let's move to **Session 14: TypeScript Patterns for Dynamic Programming (DP) & Recursion**.

This is one of the most powerful sessions — you'll learn how to use **TypeScript's type system** to make **DP and recursion problems** cleaner, faster, and bug-free.

🧭 Session 14 Overview

You'll learn:

1. The **core idea of dynamic programming** and recursion
2. How to type recursive functions properly
3. How to design **memoized** recursive functions
4. How to model **DP states** with `Record`, `Map`, and tuples
5. Examples of real **LeetCode-style problems** with typed solutions

---

🔷 1. Recursion Basics in TypeScript

Example: Factorial (Simple Recursion)

```
tsCopy codefunction factorial(n: number): number {
  if (n <= 1) return 1;
  return n * factorial(n - 1);
}
```

🧠 Type Insight:

- The function is typed as `(n: number) => number`.

- You cannot accidentally return or accept a wrong type.

✅ **Exercise:**

Implement a recursive function `sumArray(nums: number[]): number` that returns the total sum of an array.

🔷 2. Top-Down DP (Memoization)

We'll improve recursion using **memoization** to avoid recomputation.

Example: Fibonacci with Memoization

```
tsCopy codefunction fib(n: number, memo: Record<number, number> = {}): n
umber {
  if (n <= 1) return n;
  if (memo[n]) return memo[n];
  memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
  return memo[n];
}
```

🧠 **Typing Insight:**

- `Record<number, number>` ensures the memo table has numeric keys and values.
- This is **type-safe** and more readable than `any`.

✅ **Exercise:**

Implement a **memoized factorial** using `Record<number, number>`.

🔷 3. Bottom-Up DP (Iterative)

Sometimes recursion is inefficient; **bottom-up DP** avoids call stack overhead.

Example: Climbing Stairs (LeetCode #70)

```
tsCopy codefunction climbStairs(n: number): number {
  const dp: number[] = [1, 1]; // Base cases
  for (let i = 2; i <= n; i++) {
```

```
    dp[i] = dp[i - 1] + dp[i - 2];
  }
  return dp[n];
}
```

🧠 **Typing Benefit:**

- The array `dp` is explicitly `number[]`.
- TypeScript catches invalid index usage or mixed-type entries.

✅ **Exercise:**

Try converting this into a **top-down recursive version** using memoization.

🔷 4. Using Tuples and Maps for Multi-State DP

Example: Grid Traveler (LeetCode #62)

Count the number of paths in an `m x n` grid — you can move only **down** or **right**.

```
tsCopy codefunction gridTraveler(
  m: number,
  n: number,
  memo: Map<string, number> = new Map()
): number {
  const key = `${m},${n}`;
  if (memo.has(key)) return memo.get(key)!;
  if (m === 1 && n === 1) return 1;
  if (m === 0 || n === 0) return 0;

  const result =
    gridTraveler(m - 1, n, memo) + gridTraveler(m, n - 1, memo);

  memo.set(key, result);
  return result;
}
```

🧠 **Typing Insight:**

- `Map<string, number>` allows flexible keys (`"m,n"`) while maintaining type safety.
- TypeScript ensures all memo values are numbers.

✅ **Exercise:**

Implement `uniquePaths(m: number, n: number)` using the **bottom-up DP** approach.

🔷 5. DP with State Definition Using Objects

Let's solve **House Robber (LeetCode #198)** — a DP problem with "include/exclude" states.

```
tsCopy codefunction rob(nums: number[]): number {
  if (nums.length === 0) return 0;
  if (nums.length === 1) return nums[0];

  const dp: number[] = [];
  dp[0] = nums[0];
  dp[1] = Math.max(nums[0], nums[1]);

  for (let i = 2; i < nums.length; i++) {
    dp[i] = Math.max(dp[i - 1], dp[i - 2] + nums[i]);
  }

  return dp[nums.length - 1];
}
```

🧠 **Typing Benefit:**

- The `number[]` DP array guarantees numeric transitions only.
- You can't store a string or undefined value in DP accidentally.

✅ **Exercise:**

Try writing a recursive + memoized version of `rob(nums: number[])`.

🔷 6. Multi-Dimensional DP Example

Example: Knapsack Problem (0/1)

```
tsCopy codefunction knapsack(
  weights: number[],
  values: number[],
  capacity: number
): number {
  const n = weights.length;
  const dp: number[][] = Array.from({ length: n + 1 }, () ⇒
    Array(capacity + 1).fill(0)
  );

  for (let i = 1; i <= n; i++) {
    for (let w = 0; w <= capacity; w++) {
      if (weights[i - 1] <= w) {
        dp[i][w] = Math.max(
          dp[i - 1][w],
          dp[i - 1][w - weights[i - 1]] + values[i - 1]
        );
      } else {
        dp[i][w] = dp[i - 1][w];
      }
    }
  }

  return dp[n][capacity];
}
```

🧠 **Typing Advantage:**

- `number[][]` ensures proper numeric operations.
- TypeScript flags any access like `dp[i]['a']` or wrong array dimensions.

✅ **Exercise:**

Implement a **memoized recursive** version of Knapsack using `Map<string, number>`.

🔷 7. Generic Memoization Utility

You can build a **reusable memoization wrapper** in TypeScript using generics:

```
tsCopy codefunction memoize<T extends (...args: any[]) => any>(fn: T): T {
  const cache = new Map<string, ReturnType<T>>();
  return ((...args: Parameters<T>) => {
    const key = JSON.stringify(args);
    if (cache.has(key)) return cache.get(key)!;
    const result = fn(...args);
    cache.set(key, result);
    return result;
  }) as T;
}

// Example usage
const fibMemo = memoize((n: number): number => {
  if (n <= 1) return n;
  return fibMemo(n - 1) + fibMemo(n - 2);
});

console.log(fibMemo(10)); // 55
```

🧠 Typing Explanation:

- `Parameters<T>` → extracts argument types from `fn`
- `ReturnType<T>` → extracts the return type
- You can now memoize **any** function while keeping type safety.

✅ Exercise:

Use this `memoize` utility to optimize your earlier `gridTraveler()` function.

🧠 Session Summary

You learned:

- ✅ How to type recursive and DP functions
- ✅ How to use `Record`, `Map`, and tuples for state representation
- ✅ How to structure memoization in a reusable, generic way
- ✅ How to apply TypeScript's `ReturnType` and `Parameters` utilities in problem solving

---

Next, we'll move into **Session 15: TypeScript for Advanced LeetCode Challenges**, where you'll use **all these modern patterns** to solve full problems like:

- Word Break (DP + Set)
- Coin Change (DP + loops)
- Serialize/Deserialize Binary Tree (Recursion + Queue)
- Course Schedule (Graph + BFS/DFS)

Would you like to continue to **Session 15**?

You said:

yes move next to session 15

ChatGPT said:

Perfect ⚡ Let's level up to **Session 15: TypeScript for Advanced LeetCode Challenges** — this is where everything comes together.

You'll now use **real LeetCode-level problems** to see how **TypeScript types** make algorithms more robust, readable, and error-proof.

🧭 Session 15 Overview

In this session you'll:

1. Combine recursion, DP, graphs, and advanced typing
2. Write complete **LeetCode-ready TypeScript solutions**
3. Learn best practices for strong typing during problem solving
4. Solve **4 major patterns**: DP, Graphs, Recursion, and Serialization

---

🔷 1. Word Break (LeetCode #139)

> Dynamic Programming + Set + String handling

Problem:

Given a string `s` and a dictionary of words `wordDict`, return `true` if `s` can be segmented into words in the dictionary.

Solution (Bottom-Up DP)

```
tsCopy codefunction wordBreak(s: string, wordDict: string[]): boolean {
  const wordSet: Set<string> = new Set(wordDict);
  const dp: boolean[] = Array(s.length + 1).fill(false);
  dp[0] = true;

  for (let i = 1; i <= s.length; i++) {
    for (let j = 0; j < i; j++) {
      const word = s.slice(j, i);
      if (dp[j] && wordSet.has(word)) {
```

```
      dp[i] = true;
      break;
     }
    }
   }

   return dp[s.length];
 }
```

🧠 **Typing Benefits:**

- `Set<string>` prevents duplicate word entries and ensures only strings are stored.
- `boolean[]` guarantees valid truthy/falsy states — no undefined indexes.

✅ **Exercise:**

Modify this to also **return one valid segmentation path** (e.g., `["leet", "code"]`).

🔷 2. Coin Change (LeetCode #322)

> Bottom-Up DP for minimum coins

Problem:

Given coins of different denominations, find the minimum number needed to make up a given amount.

Solution

```
tsCopy codefunction coinChange(coins: number[], amount: number): number
{
 const dp: number[] = Array(amount + 1).fill(Infinity);
 dp[0] = 0;

 for (const coin of coins) {
  for (let i = coin; i <= amount; i++) {
   dp[i] = Math.min(dp[i], dp[i - coin] + 1);
  }
```

```
  }

  return dp[amount] === Infinity ? -1 : dp[amount];
}
```

🧠 **Typing Benefits:**

- `number[]` ensures consistent numeric DP state.
- TypeScript prevents logical errors like comparing strings or booleans.

✅ **Exercise:**

Return **both the minimum number and the combination** of coins that achieve it.

🔷 3. Serialize and Deserialize Binary Tree (LeetCode #297)

> Tree traversal + recursion + queue

Tree Node Definition

```
tsCopy codeclass TreeNode {
  val: number;
  left: TreeNode | null;
  right: TreeNode | null;
  constructor(val: number, left: TreeNode | null = null, right: TreeNode | null = n
ull) {
    this.val = val;
    this.left = left;
    this.right = right;
  }
}
```

Solution

```
tsCopy codefunction serialize(root: TreeNode | null): string {
  const result: (number | "null")[] = [];

  const dfs = (node: TreeNode | null): void => {
    if (!node) {
      result.push("null");
      return;
    }
    result.push(node.val);
    dfs(node.left);
    dfs(node.right);
  };

  dfs(root);
  return result.join(",");
}

function deserialize(data: string): TreeNode | null {
  const values = data.split(",");

  const buildTree = (): TreeNode | null => {
    const val = values.shift();
    if (val === "null" || val === undefined) return null;
    const node = new TreeNode(Number(val));
    node.left = buildTree();
    node.right = buildTree();
    return node;
  };

  return buildTree();
}
```

🧠 **Typing Benefits:**

- `TreeNode | null` ensures recursive consistency — no undefined errors.

- TS enforces return type safety for both serialize/deserialize.

✅ **Exercise:**

Add a `printInorder(root: TreeNode | null)` function for validation.

🔷 4. Course Schedule (LeetCode #207)

> Graph + BFS (Topological Sort)

Problem:

Given `numCourses` and a list of prerequisites, determine if it's possible to finish all courses.

Solution

```
tsCopy codefunction canFinish(numCourses: number, prerequisites: number[][]): boolean {
  const graph: Map<number, number[]> = new Map();
  const indegree: number[] = Array(numCourses).fill(0);

  for (const [a, b] of prerequisites) {
    if (!graph.has(b)) graph.set(b, []);
    graph.get(b)!.push(a);
    indegree[a]++;
  }

  const queue: number[] = [];
  for (let i = 0; i < numCourses; i++) {
    if (indegree[i] === 0) queue.push(i);
  }

  let count = 0;
  while (queue.length) {
    const course = queue.shift()!;
    count++;
    for (const next of graph.get(course) || []) {
```

```
      indegree[next]--;
      if (indegree[next] === 0) queue.push(next);
    }
  }

  return count === numCourses;
}
```

🧠 **Typing Highlights:**

- `Map<number, number[]>` → strictly typed adjacency list.
- Ensures no invalid course index or structure error.

✅ **Exercise:**

Modify this to return the **actual topological order** of courses if possible.

🔷 5. Combine Everything: TypeScript Patterns Summary

Pattern	TypeScript Strength	Example Problem
DP (Array-based)	Strongly typed state transitions	Coin Change
Recursion + Memo	Generic memoization, Record<Map> types	Fibonacci, Grid Traveler
Graph	`Map<T, Set<T>>` guarantees consistent structure	Course Schedule
Tree	`TreeNode	null` prevents runtime errors
String DP	`boolean[]` and `Set<string>` safety	Word Break

🧠 Session Summary

You learned how to:

- ✅ Build **LeetCode-level solutions** with strong typing
- ✅ Combine multiple TypeScript features in algorithmic problems
- ✅ Use `Map`, `Set`, `Record`, `Tuple`, and classes in real contexts
- ✅ Prevent common JS runtime errors using **TypeScript's type system**

Next, we move to the **final session (Session 16): Building Your Own TypeScript LeetCode Template & Helper Library**,

where you'll design a reusable **TypeScript problem-solving environment** with:

- Custom input/output helpers
- Typed DP and graph utilities
- Reusable data structure templates

Would you like me to continue to **Session 16 (Final Session)**?

You said:

yes move to final session (Thank God)

ChatGPT said:

😄 Awesome! Let's wrap it all up strong.

🏁 Final Session — Session 16: Mastering TypeScript for LeetCode Problem Solving

🎯 Goal

To consolidate everything you've learned — modern types, algorithmic patterns, and LeetCode workflows — into a *repeatable, powerful problem-solving framework* using TypeScript.

---

1. Review: What You've Mastered So Far

Concept	You Can Now...
Objects & Tuples	Define structured data and maintain type safety.
Maps, Sets, Records	Store and query dynamic data with predictable types.
Unions & Generics	Write reusable, type-safe logic adaptable to any input.
Utility Types	Transform existing types (`Partial` , `Pick` , `Omit` , etc.) efficiently.
Algorithmic Patterns	Express common LeetCode solutions idiomatically in TypeScript.
Data Structures	Implement linked lists, stacks, queues, trees, graphs, and heaps.

---

2. TypeScript + LeetCode Workflow

Here's a streamlined workflow for using TS effectively on problem-solving platforms:

1. **Read the problem carefully.**
    - Extract constraints and input/output structure.
    - Immediately define a `type` or `interface` for inputs.
2. **Prototype with Types.**

- Let TypeScript *guide* you — define function signatures first.
- Example:

```
tsCopy codetype TwoSumFn = (nums: number[], target: number) => number[];
```

3. **Code, then Refactor with Modern Types.**
   - Replace loose typing (`any`) with generics or Record types.
   - Use maps/sets when you detect repeated lookups.
4. **Test with Edge Cases.**
   - TypeScript helps surface logical edge cases (e.g., undefined or empty arrays).
5. **Benchmark for Performance.**
   - Types don't affect runtime, but type clarity helps you *think* efficiently.

---

3. Example: Bringing It All Together

🧩 Problem — "Group Anagrams"

```
tsCopy codefunction groupAnagrams(strs: string[]): string[][] {
  const map: Map<string, string[]> = new Map();

  for (const word of strs) {
    const key = word.split('').sort().join('');
    if (!map.has(key)) map.set(key, []);
    map.get(key)!.push(word);
  }

  return Array.from(map.values());
}
```

✅ Modern TypeScript Concepts Used

- `Map<K, V>`
- Non-null assertion (`!`)
- Array typing (`string[][]`)
- Inferred return type safety

---

4. TypeScript Problem-Solving Framework

Phase	Step	TypeScript Concept
Understand	Define input/output types	`type` , `interface`
Plan	Think in data structures	`Map` , `Set` , `Record`
Implement	Write algorithm	Generics, tuples, unions
Refactor	Make reusable	Utility types, inference
Validate	Test with edge cases	Type guards, assertions

---

5. Advanced Exercise (Final Project)

🔷 **Goal:** Implement a type-safe `LRUCache<K, V>` class in TypeScript.

Requirements:

- Use `Map<K, V>` internally.
- Maintain a fixed capacity.
- When capacity exceeds, remove the least recently used key.
- Ensure the entire class is *fully generic*.

Example:

```
tsCopy codeclass LRUCache<K, V> {
  private capacity: number;
  private cache: Map<K, V>;
```

```
  constructor(capacity: number) {
    this.capacity = capacity;
    this.cache = new Map();
  }

  get(key: K): V | undefined {
    if (!this.cache.has(key)) return undefined;
    const value = this.cache.get(key)!;
    this.cache.delete(key);
    this.cache.set(key, value);
    return value;
  }

  put(key: K, value: V): void {
    if (this.cache.has(key)) this.cache.delete(key);
    else if (this.cache.size >= this.capacity)
      this.cache.delete(this.cache.keys().next().value);
    this.cache.set(key, value);
  }
}
```

🎯 **Concepts Reinforced:**

- Generics `<K, V>`
- Map iteration
- Encapsulation
- Algorithmic logic + types

---

6. What's Next

🎓 **You're now ready to:**

- Solve any LeetCode problem using **TypeScript confidently**.
- Build reusable, type-safe algorithm templates.
- Think like a **TypeScript engineer**, not just a coder.

🧠 **Next Steps (Recommended Path):**

1. Solve 2–3 problems daily using TypeScript.
2. Focus on using new types in each.
3. Gradually move to advanced generics and type-level programming.

Would you like me to generate a **final "TypeScript Problem Solving Cheatsheet"** (a one-page summary of all modern types + LeetCode patterns) as a bonus after this session?

You said:

Yes you should generate before going, happy day

ChatGPT said:

😄 Perfect — let's end this journey with your **TypeScript Problem Solving Cheatsheet** — your go-to quick reference for LeetCode and beyond 🚀

🧩 TypeScript Problem Solving Cheatsheet

🎯 Goal

A compact reference of **modern TypeScript types, data structures, and patterns** you've mastered — everything you need to code confidently on LeetCode.

🧱 Core Data Types

Category	Example	Notes
Primitives	`number`, `string`, `boolean`, `null`, `undefined`	Basic building blocks
Array	`number[]`, `Array<string>`	Homogeneous lists
Tuple	`[number, string]`	Fixed-size, mixed types
Object	`{ name: string; age: number }`	Custom structured type
Enum	`enum Direction { Up, Down }`	Readable constants
Union	`string	number`
Literal	`"on"	"off"`
Intersection	`A & B`	Combine multiple types

🧩 Advanced Data Structures

Structure	Type	Use Case
Map	`Map<K, V>`	Key-value with any key type
Set	`Set<T>`	Unique values (no duplicates)
Record	`Record<K, V>`	Typed dictionary with fixed keys
WeakMap / WeakSet	`WeakMap<object, any>`	Memory-safe references
ReadonlyMap / ReadonlyArray	—	Immutable variants

🧠 **Tip:** Use `Map` when you need O(1) lookups, `Set` for fast membership checks, and `Record` for static key-value typing.

🧩 Type Utilities

Utility	Description	Example
`Partial<T>`	All fields optional	`Partial<User>`
`Required<T>`	All fields required	`Required<User>`
`Pick<T, K>`	Pick certain fields	`Pick<User, 'id'
`Omit<T, K>`	Remove certain fields	`Omit<User, 'password'>`
`Readonly<T>`	Make all fields readonly	`Readonly<User>`
`Record<K, T>`	Object with specific key-value types	`Record<string, number>`
`ReturnType<T>`	Extract function’s return type	`ReturnType<typeof fn>`
`Parameters<T>`	Extract function’s parameter types	`Parameters<typeof fn>`

⚙️ Generics

🔷 **Why:** Make reusable, flexible code that adapts to multiple data types.

```
tsCopy codefunction identity<T>(value: T): T {
  return value;
}
```

🔷 **Generic Classes:**

```
tsCopy codeclass Box<T> {
  constructor(public content: T) {}
}
```

🔷 **Generic Constraints:**

```
tsCopy codefunction logLength<T extends { length: number }>(item: T) {
  console.log(item.length);
}
```

🧠 Type Guards

```
tsCopy codefunction isString(value: unknown): value is string {
  return typeof value === "string";
}
```

✅ Helps the compiler narrow types during runtime checks.

🧩 Common Patterns in LeetCode

Pattern	TypeScript Concepts Used	Example Problem
Two-Pointer	Arrays, tuples	Move Zeroes, Container With Most Water
Sliding Window	Maps, sets	Longest Substring Without Repeating
Hash Map Counting	`Map<K,V>`	Two Sum, Anagrams
Dynamic Programming	Arrays, memoization with `Record<number, number>`	Climbing Stairs
Graph Traversal (DFS/BFS)	Generics, recursion, sets	Number of Islands
Heap / Priority Queue	Class + Generics	Kth Largest Element
Backtracking	Arrays + recursion	Combination Sum

🧩 Performance Notes

Action	Recommended TypeScript Practice
Frequent lookups	`Map` instead of `{}`
Unique elements	`Set`
Type safety with APIs	`interface` or `type`

Action	Recommended TypeScript Practice
Immutable results	`ReadonlyArray` or `as const`
Optional parameters	`?` or `Partial<>`
Safer narrowing	Type guards (`value is Type`)

🧩 TypeScript LeetCode Setup Tip

LeetCode supports TypeScript (ts) directly:

```
tsCopy codefunction twoSum(nums: number[], target: number): number[] {
  const map = new Map<number, number>();
  for (let i = 0; i < nums.length; i++) {
    const complement = target - nums[i];
    if (map.has(complement)) return [map.get(complement)!, i];
    map.set(nums[i], i);
  }
  return [];
}
```

✅ Clean, readable, and 100% type-safe.

🏁 Final Wisdom

> "TypeScript isn't about types — it's about clarity.
>
> The clearer your types, the fewer your bugs."

🧭 What You Can Do Next

1. Try 3 LeetCode problems daily — in **TypeScript only**.
2. For every solution, focus on **using one new type** you learned.
3. Build your own **Type Utility Library** for patterns you repeat often.

Would you like me to generate a **printable PDF version** of this cheatsheet (formatted nicely for reference while coding)?

ChatGPT can make mistakes. Check important info.