Topic 3: Advanced Types (Union, Intersection, Mapped, Conditional, Utility Types)

1. Problem Statement

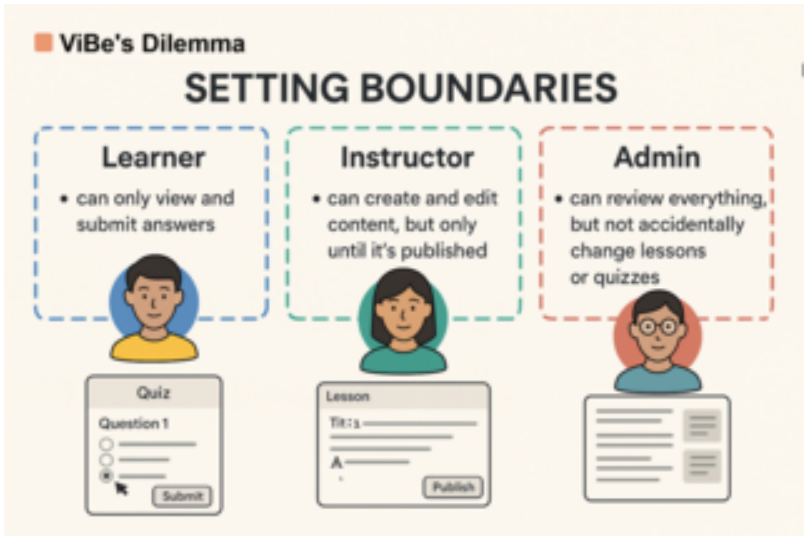
ViBe’s Dynamic Dashboard Dilemma

ViBe’s dashboard must show:

- Each learner’s progress (quizzes, videos, assignments, etc.)
- Each instructor’s engagement (courses taught, feedback given)
- Each admin’s reports (user stats, revenue, system alerts)

But the data is always changing:

- Some learners do only quizzes, others only videos.
- Some instructors are also learners.
- Admins want to combine, filter, and transform data for custom reports.



The problem:
How do you build a system that can:

- Combine different data shapes safely?
- Handle users with multiple roles?
- Transform or filter data for any report, without rewriting code for every new dashboard feature?

Expected outcome:

- Flexible, type-safe code that adapts to real-world data complexity.
- Reports and features that are easy to extend, without bugs or duplication.

2. Learning Objectives

By the end of this lesson, you’ll be able to:

- Use union and intersection types to model flexible data.
- Use mapped types to transform or generate new data shapes.
- Use conditional types to make types adapt to situations.

- Use utility types to quickly adjust existing types.
- Build robust, future-proof code for ViBe's dynamic needs.

3. Concept Introduction with Analogy

The Swiss Army Knife Analogy

Imagine your code is a Swiss Army knife:

- Sometimes you need just a knife (one type).
- Sometimes you need a knife and a screwdriver at once (combine types).
- Sometimes you want to quickly reshape a tool for a special job (transform types).
- Sometimes you want a tool that adapts itself based on the job (conditional types).

Advanced types let you build tools that are just as flexible and adaptable as your real-world needs.

Union Types

A union type allows a value to be **one of several types**.
It's written with the `|` ("or") operator.

How They Work

- When a value is a union, TypeScript only allows you to use members/methods that are common to all types in the union.
- To use type-specific properties, you must **narrow** the type (using `typeof`, `in`, or discriminant properties).

```
type StringOrNumber = string | number;
let value: StringOrNumber;
value = "hello"; // OK
value = 42;      // OK
value = true;    // Error
```

Intersection Types

An intersection type combines **multiple types into one** using the `&` ("and") operator.

How They Work

- The resulting type has **all properties** of the intersected types.
- If there's a property with the same name but different types, TypeScript will error.

```
type A = { a: number };
type B = { b: string };
type AB = A & B; // { a: number; b: string }
```

```
const obj: AB = { a: 1, b: "hi" }; // OK
```

Mapped Types

Mapped types let you **transform each property** in a type, often used with `keyof` and unions.

How They Work

- You “map over” the keys of a type or union and create a new type for each key.
- Used for making all properties optional, readonly, or changing their types.

```
type User = { id: string; name: string; };
type OptionalUser = { [K in keyof User]?: User[K] }; // All properties optional
```

Conditional Types

What Are They?

Conditional types let you **choose a type based on a condition** at the type level, using the form `T extends U ? X : Y`.

- TypeScript checks if the type on the left of `extends` is assignable to the one on the right.
- If true, the first branch is used; otherwise, the second.

```
type IsString<T> = T extends string ? true : false;

type A = IsString<string>; // true
type B = IsString<number>; // false
```

4. Step-by-Step Data Modeling

Let’s model ViBe’s users and reports:

```
// Basic user types
type Learner = { id: string; quizzesCompleted: number };
type Instructor = { id: string; coursesTaught: number };
type Admin = { id: string; accessLevel: "basic" | "super" };

// Some users have more than one role!
type MultiRoleUser = Learner & Instructor; // Intersection: both!
type AnyUser = Learner | Instructor | Admin; // Union: one or the other
```

5. Live Code Walkthrough (Step-by-Step, Explained)

A. Union Types: One or the Other

```
function printUserInfo(user: AnyUser) {
 if ("quizzesCompleted" in user) {
```

```
    console.log(`Learner: ${user.quizzesCompleted} quizzes completed`);
  } else if ("coursesTaught" in user) {
    console.log(`Instructor: ${user.coursesTaught} courses taught`);
  } else {
    console.log(`Admin: Access - ${user.accessLevel}`);
  }
}
```

- **Union types** let you accept different shapes and handle each safely.

B. Intersection Types: Both at Once

```
const alice: MultiRoleUser = {
  id: "alice123",
  quizzesCompleted: 10,
  coursesTaught: 2
};
```

- **Intersection types** let you combine all properties for users with multiple roles.

C. Mapped Types: Transform Data Shapes

Suppose you want to track module progress for each learner:

```
type ModuleStatus = "not-started" | "in-progress" | "completed";

// For any list of module IDs, generate a progress map:
type ProgressMap<Modules extends string> = {
  [K in Modules]: ModuleStatus;
};

type MyModules = "quiz1" | "video2" | "assignment3";
type MyProgress = ProgressMap<MyModules>;

// Result: { quiz1: ModuleStatus; video2: ModuleStatus; assignment3: ModuleStatus }
```

D. Conditional Types: Adapting to Data

Suppose you want to allow feedback only if a module is completed:

```
type FeedbackAllowed<T extends ModuleStatus> = T extends "completed" ? string : never;

type FeedbackForQuiz = FeedbackAllowed<"completed">; // string
type FeedbackForVideo = FeedbackAllowed<"in-progress">; // never
```

E. Utility Types: Quick Type Transformations

Suppose you want a type where all fields are optional for a draft report:

```
type LearnerReport = {
  name: string;
  score: number;
  feedback: string;
};
type DraftReport = Partial<LearnerReport>; // All fields now optional
```

Or make a type where all fields are read-only:

```
type ReadonlyReport = Readonly<LearnerReport>;
```

6. Challenge

Your Turn!

1. Create a `type` called `InstructorOrAdmin` that can be either an `Instructor` or an `Admin`.
2. Given a type `Assignment = { title: string; dueDate: Date; points: number; }`, create a type `ReadonlyAssignment` where none of the fields can be changed.
3. Given a type `LearnerStats = { quizzes: number; videos: number; assignments: number; }`, create a type `StatsAsStrings` that has the same keys, but all values are strings.

7. Quick Recap & Key Takeaways

- **Unions:** When a value can be one of several types (e.g., API responses, user roles).
- **Intersections:** When you need a type that combines multiple behaviors or data sources.
- **Mapped Types:** When you want to transform or generate types based on existing ones (e.g., make all fields optional).
- **Conditional Types:** For type-level logic and type transformations (e.g., extracting types, enforcing rules).
- **Utility Types:** For common type manipulations and code reuse.

8. (Optional) Programmer's Workflow Checklist

- Use **union** when a value can be one of many types.
- Use **intersection** when you need all properties from multiple types.
- Use **mapped** and **utility types** to quickly reshape data.
- Use **conditional types** for smart, rules-based types.
- Test your types with real data and scenarios.

9. Coming up Next

Congratulations!
You've mastered the Swiss Army knife of TypeScript types.
Next, imagine combining all these tools to build a reporting engine that adapts to any new content or user type ViBe invents in the future!