Conditional Types in TypeScript

Enhancing Type Flexibility and Expressiveness

What are Conditional Types?

- A way to create type definitions based on conditions
- Similar to ternary operators in JavaScript
- Allow for type-level programming
- Introduced in TypeScript 2.8
- Enable creation of more flexible and reusable type definitions

Basic Syntax of Conditional Types

- Uses the form: `T extends U ? X : Y`
- `T extends U` is the condition
- If `T` is assignable to `U`, the type resolves to `X`
- Otherwise, it resolves to `Y`
- Can be nested for more complex conditions
- Often used with generics for maximum flexibility

```
type NumberOrString<T> = T extends number ? number : string;

let num: NumberOrString<number> = 42;
let str: NumberOrString<boolean> = "Hello";

// This would cause a type error:
let error: NumberOrString<number> = "This will fail";
```

Real-World Example: NonNullable

- `NonNullable<T>` is a built-in conditional type
- Removes `null` and `undefined` from a type `T`
- Useful for ensuring non-nullable types
- Demonstrates the practical application of conditional types
- Can be used to create more robust function parameters and return types

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

type MaybeString = string | null | undefined;
type DefinitelyString = NonNullable<MaybeString>; // Type is string

function processValue<T>(value: NonNullable<T>) {
    // We can safely use 'value' here, knowing it's not null or undefined
    console.log(value);
}

processValue("Hello");
processValue(42);
processValue(null);
processValue(undefined);
```

Advanced Usage: Inferring Within Conditional Types

- Use `infer` keyword to infer types within a conditional type
- Allows for extracting types from complex structures
- Useful for creating utility types
- Can infer return types, parameter types, and more
- Enables powerful type transformations

```
type ReturnType<T> = T extends (...args: any[]) => infer R ? R : any;

function greet(name: string): string {
    return `Hello, ${name}!`;
}

type GreetReturn = ReturnType<typeof greet>;

const numberArray = [1, 2, 3, 4, 5];
type ArrayElement<T> = T extends (infer U)[] ? U : never;
type NumberArrayElement = ArrayElement<typeof numberArray>;
```

Practical Application: Discriminated Unions 1/2

- Conditional types work well with discriminated unions
- Can create type-safe functions that return different types based on input
- Useful for handling different shapes of data with type-specific operations
- Enhances type safety and developer experience in complex data structures
- Enables precise type inference in function returns

Practical Application: Discriminated Unions 2/2

```ts
type Circle = { kind: "circle"; radius: number };
type Rectangle = { kind: "rectangle"; width: number; height: number };
type Square = { kind: "square"; sideLength: number };

type Shape = Circle | Rectangle | Square;

type ShapeInfo<T extends Shape> = T extends Circle
    ? { area: number; circumference: number }
    : T extends Rectangle
      ? { area: number; perimeter: number }
      : T extends Square
        ? { area: number; diagonal: number }
        : never;

function getShapeInfo<T extends Shape>(shape: T): ShapeInfo<T> {
  switch (shape.kind) {
    case "circle":
      const area = Math.PI * shape.radius ** 2;
      const circumference = 2 * Math.PI * shape.radius;
      return { area, circumference } as ShapeInfo<T>;
    case "rectangle":
      const rectArea = shape.width * shape.height;
      const perimeter = 2 * (shape.width + shape.height);
      return { area: rectArea, perimeter } as ShapeInfo<T>;
    case "square":
      const squareArea = shape.sideLength ** 2;
```

Benefits of Conditional Types

- Enhanced type flexibility and expressiveness
- Ability to create more precise and reusable utility types
- Improved type inference in complex scenarios
- Enables type-level programming and metaprogramming

Best Practices and Considerations

- Use Conditional Types judiciously to avoid overly complex type definitions
- Leverage existing utility types before creating new ones
- Consider readability and maintainability when creating complex type structures
- Use meaningful names for type parameters and conditional types
- Document complex Conditional Types for easier understanding
- Be aware of potential performance impacts with very complex type computations

Questions?

Have you used Conditional Types in your TypeScript projects?

Thank You!

Remember: Conditional Types are a powerful tool, use them wisely to create more expressive and type-safe code!