Indexed Types in TypeScript

Accessing and Extracting Types Like Object Properties

Understanding Indexed Types

- Access properties of types using square brackets
- Similar to accessing object properties in JavaScript
- Powerful way to extract and manipulate types
- Foundation for many advanced TypeScript patterns

Basic Syntax: Object Properties

- Use square bracket notation to access type properties
- Can access any property that exists on the type
- Useful for extracting specific property types
- Type-safe: TypeScript ensures property exists

```
type NavbarProps = {
  onChange: () => void;
  title: string;
  isVisible: boolean;
};

// Extract specific property types
type OnChangeType = NavbarProps["onChange"];
type TitleType = NavbarProps["title"];

// Can even access multiple properties using union
type StringProps = NavbarProps[keyof NavbarProps];

type Invalid = NavbarProps["notExists"];
```

Array Types with T[number]

- Special syntax for accessing array element types
- `T[number]` extracts type of array elements
- Commonly used with `typeof` and `as const`
- Powerful way to derive types from array values

```
// Basic array type extraction
const numbers = [1, 2, 3, 4, 5]; // add a string
type NumberArray = typeof numbers;
type NumberType = NumberArray[number];

// With 'as const' for literal types
const roles = ["admin", "editor", "contributor"] //as const;
type RolesArray = typeof roles;
type Role = RolesArray[number];
```

Advanced Usage: Numeric Index Signatures

- Access types defined by numeric index signatures
- Useful for array-like structures
- Can coexist with regular properties

```
type ExampleObj = {
  // String keys
  stringKey1: "string-value";
  stringKey2: "string-value";
  [key: number]: "number-value";
};
type StringKeyType = ExampleObj["stringKey1"];

// Practical example with array-like structure
type ArrayLike<T> = {
  length: number;
  [index: number]: T;
  map(fn: (item: T) => any): any[];
};

type ArrayElement = ArrayLike<string>[number];    // string

export {}
```

Real-World Applications

- Creating type-safe enums from arrays
- Deriving types from test fixtures
- Extracting types from configuration objects
- Building flexible library interfaces
- Maintaining type synchronization with data

```typescript
// Type-safe configuration object
const config = {
  api: {
    endpoint: "https://api.example.com",
    timeout: 5000,
    retries: 3,
  },
  features: {
    darkMode: true,
    betaFeatures: false,
  },
  theme: {
    primary: "#007AFF",
    secondary: "#5856D6",
  },
} as const;

// Helper type to remove readonly modifiers and widen literal types
type Mutable<T> = {
  -readonly [P in keyof T]: T[P] extends object
    ? Mutable<T[P]>
    : T[P] extends string
      ? string
      : T[P] extends number
        ? number
        : T[P] extends boolean
          ? boolean
          : T[P];
};
```

Benefits and Best Practices

- DRY principle in type definitions
- Type safety without manual maintenance
- Improved refactoring capabilities
- Reduced chance of type mismatches

Questions?

Have you used indexed types in your TypeScript projects?

Thank You!

Remember: Indexed types are a powerful way to keep your types DRY and in sync with your data!