# Type Aliases vs. Interfaces in TypeScript

Embracing the Power and Flexibility of Type Aliases

# Type Aliases: Your Swiss Army Knife

- Can define any type, not just objects
- Support unions, intersections, and mapped types
- More flexible and expressive
- Consistent syntax for all types

```
// Object type
type User = {
  name: string;
  age: number;
};

// Union type
type ID = string | number | null;

// Function type
type Callback = (url: string) => void

// Intersection type
type Draggable = { drag: () => void };
type Resizable = { resize: () => void };
type UIElement = Draggable & Resizable;

// Mapped type
type Nullable<T> = { [P in keyof T]: T[P] | null };
type UpdateUser = Nullable<User>;

// Conditional type
type NonNullable<T> = T extends null | undefined ? never : T;
type NonNullableID = NonNullable<ID>
```

# When to Use Type Aliases vs. Interfaces

## Type Aliases (Preferred)

- For most type definitions
- When working with unions or intersections
- For complex type manipulations
- When you need aliases for primitives

## Interfaces (Specific Use Cases)

- When you need declaration merging
- For defining class shapes (implements)
- When extending built-in or third-party interfaces

# The Few Cases for Interfaces (1/2)

```ts
// Declaration merging
interface Window {
  customProperty: string;
}

// Class implementation
interface Printable {
  print(): void;
}

class Document implements Printable {
  // print() {
  //   console.log("Printing document...");
  // }
}
```

## When to Consider Interfaces

- Declaration merging (augmenting types across files)
- Defining class contracts with `implements`

# The Few Cases for Interfaces (2/2)

```
// Extending library interfaces (e.g., jQuery)
interface JQuery {
  t(key: string): string;
}

const $element = $('div');
$element.t('hello.world');

interface JQueryStatic {
  i18n: {
    t(key: string): string;
  };
}

// Usage of static method
$.i18n.t('hello.world');
```

## When to Consider Interfaces Part 2

- Extending library-defined interfaces (e.g., jQuery)

## Best Practices

- Use type aliases as your default choice
- Leverage unions and intersections for flexible types
- Use mapped and conditional types for advanced scenarios
- Reserve interfaces for specific use cases (merging, implements)
- Be consistent in your usage across the project

## Remember

- Type aliases are more versatile and powerful
- They provide a consistent syntax for all types
- Interfaces have their place, but have different features
- Choose based on the needs of your project
- When in doubt, start with a type alias!

# Questions?

Type Aliases: The unsung heroes of TypeScript?

# Thank You!

Remember: In TypeScript, type aliases unlock true power!