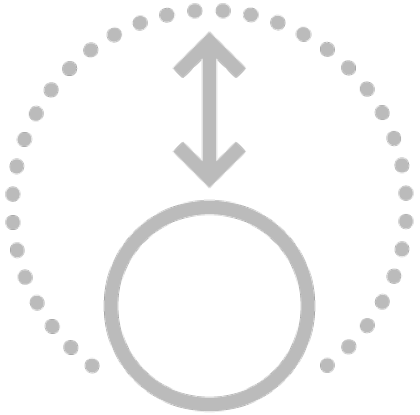
Using Interfaces to Future-Proof Code

Why Interfaces?

Maintainable



Extensible

Easily testable

Best Practices

[OPTION A]

Best Practice

Program to an abstraction rather than a concrete type

[OPTION A]

Program to an interface
rather than a concrete class

[OPTION B]

Program to an abstraction rather than a concrete type

[OPTION B]

Program to an interface rather than a concrete class

Concrete Classes

Concrete Classes

Collections

- List<T>
- Array
- SortedList<TKey, TValue>
- HashTable
- Queue / Queue<T> — FIFO
- Stack / Stack<T> — LIFO
- Dictionary<TKey, TValue>
- ObservableCollection<T>
- + Custom Types

```
Public class List<T> : Ilist<T>
       ICollection<T>, Ilist, Icollection,
       IReadOnlyList<T>, IReadOnlyCollection<T>
       IEnumerable<T>, IEnumerable
```

Collection Interfaces

Interface Segregation Principle

```
Public class List<T> : Ilist<T>
        ICollection<T>, Ilist, Icollection,
        IReadOnlyList<T>, IReadOnlyCollection<T>
        IEnumerable<T>, IEnumerable
```

IEnumerable

Used with

- foreach
- List Boxes

Summary

Best Practice

- Program to an abstraction rather than a concrete type

or

- Program to an interface rather than a concrete class

Summary

Concrete Class

- Brittle / Easily Broken

Interface

- Resilience in the face of change
- Insulation from implementation details

UP NEXT:

The “How” of Interfaces