

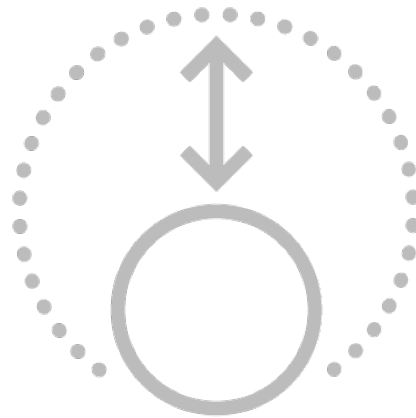
Using Interfaces to Future-Proof Code



Why Interfaces?



Maintainable



Extensible



Easily testable

Best Practices



Best Practice

Program to an abstraction
rather than a concrete type



[OPTION A]



Program to an interface
rather than a concrete class



Program to an abstraction
rather than a concrete type



Program to an interface
rather than a concrete class



Concrete Classes



Concrete Classes

Collections

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

FIFO

LIFO



```
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

