Advanced Interface Topics

Where To Go Next

Jeremy Clark
www.jeremybytes.com
jeremy@jeremybytes.com

pluralsight
hardcore developer training

Overview

- **Best Practices**
  - Interface Segregation Principle
  - Choosing Between Abstract Class and Interface
  - Updating Interfaces

- **Advanced Topics**
  - Dependency Injection
  - Mocking

Interface Segregation Principle

```
public class List<T> : IList<T>,
    ICollection<T>,IList, ICollection,
    IReadOnlyList<T>,IReadOnlyCollection<T>,
    IEnumerable<T>,IEnumerable
```

Clients should not be forced to depend upon methods that they do not use. Interfaces belong to clients, not hierarchies.*

*Martin & Martin. *Agile Principles, Patterns, and Practices in C#*. Pearson Education, 2006.

Interface Segregation Principle

```
public class List<T> : IList<T>,
    ICollection<T>,IList, ICollection,
    IReadOnlyList<T>,IReadOnlyCollection<T>,
    IEnumerable<T>,IEnumerable
```

We should have granular interfaces that only include the members that a particular function needs.

List<T> Interfaces

```
public class List<T> : IList<T>,
    ICollection<T>,IList, ICollection,
    IReadOnlyList<T>,IReadOnlyCollection<T>,
    IEnumerable<T>,IEnumerable
```

IEnumerable

GetEnumerator()

IEnumerable<T>

GetEnumerator()

List<T> Interfaces

```
public class List<T> : IList<T>,
    ICollection<T>, IList, ICollection,
    IReadOnlyList<T>, IReadOnlyCollection<T>,
    IEnumerable<T>, IEnumerable
```

ICollection<T>

Count
IsReadOnly
Add()
Clear()
Contains()
CopyTo()
Remove()

Plus,
Everything in
IEnumerable<T>,
IEnumerable

List<T> Interfaces

```
public class List<T> : IList<T>,
    ICollection<T>,IList, ICollection,
    IReadOnlyList<T>,IReadOnlyCollection<T>,
    IEnumerable<T>,IEnumerable
```

IList<T>

Item / Indexer
IndexOf()
Insert()
RemoveAt()

Plus,
Everything in
ICollection<T>,
IEnumerable<T>,
IEnumerable

Granular Interfaces

- **If We Need to**
  - Iterate over a Collection / Sequence
  - Data Bind to a List Control
  - Use LINQ functions

IEnumerable<T>

- **If We Need To**
  - Add/Remove Items in a Collection
  - Count Items in a Collection
  - Clear a Collection

ICollection<T>

- **If We Need To**
  - Control the Order Items in a Collection
  - Get an Item by the Index

IList<T>

IEnumerable Implementations

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

IEnumerable<T> Implementations

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

ICollection<T> Implementations

List<T>
SortedList<TKey, TValue>
Dictionary<TKey, TValue>
+
Custom Types

IList<T> Implementations

List<T>

+

Custom Types

Program at the Right Level

- **If We Need to**
  - Iterate over a Collection / Sequence
  - Data Bind to a List Control

IEnumerable<T>

- **If We Need To**
  - Add/Remove Items in a Collection
  - Count Items in a Collection
  - Clear a Collection

ICollection<T>

- **If We Need To**
  - Control the Order Items in a Collection
  - Get an Item by the Index

IList<T>

IPersonRepository

```
public interface IPersonRepository
{
    IEnumerable<Person> GetPeople();

    Person GetPerson(string lastName);

    void AddPerson(Person newPerson);

    void UpdatePerson(string lastName, Person updatedPerson);

    void DeletePerson(string lastName);

    void UpdatePeople(IEnumerable<Person> updatedPeople);
}
```

Better Segregation

```
public interface IReadOnlyPersonRepository
{
    IEnumerable<Person> GetPeople();

    Person GetPerson(string lastName);
}
```

```
public interface IPersonRepository : IReadOnlyPersonRepository
{
    void AddPerson(Person newPerson);

    void UpdatePerson(string lastName, Person updatedPerson);

    void DeletePerson(string lastName);

    void UpdatePeople(IEnumerable<Person> updatedPeople);
}
```

Abstract Class vs. Interface

Abstract Classes

- May contain implementation code

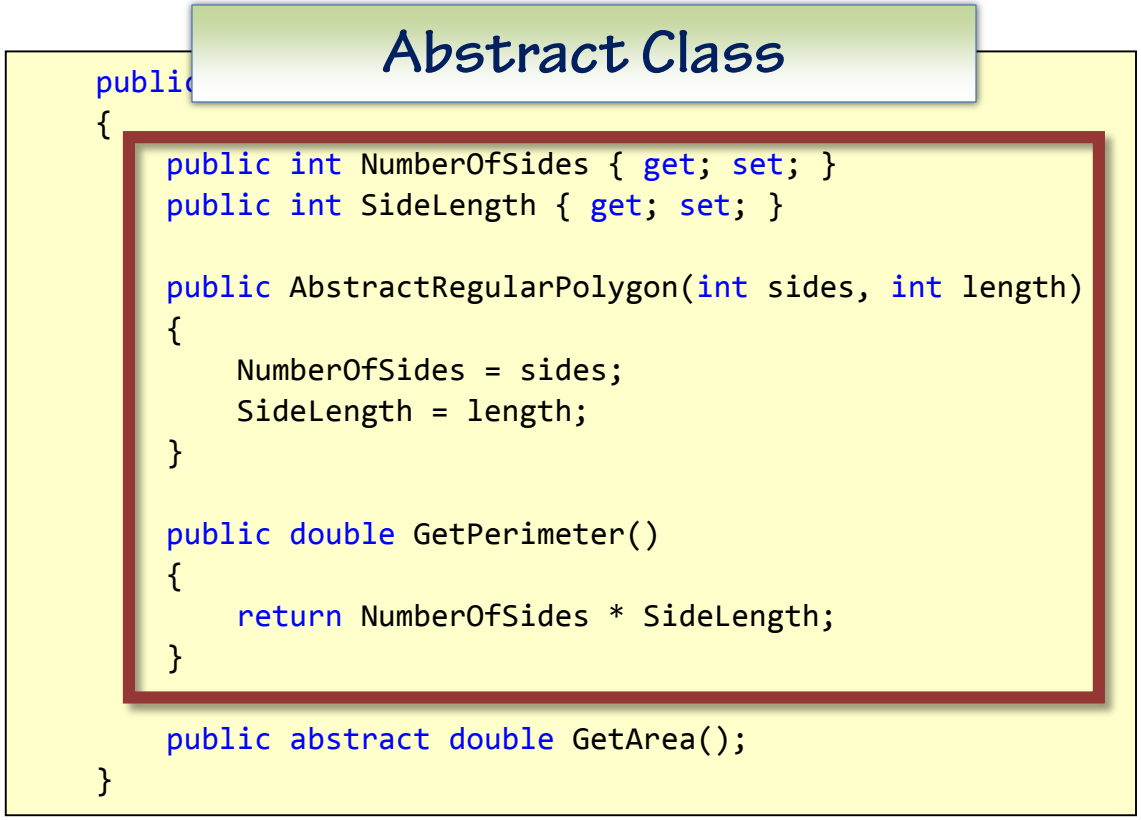
- A class may inherit from a single base class
  - Members have access modifiers
  - May contain fields, properties, constructors, destructors, methods, events and indexers

Interfaces

- May not contain implementation code

- A class may implement any number of interfaces
  - Members are automatically public
  - May only contain properties, methods, events, and indexers

Regular Polygon



Lots of Shared Code

Person Repository

Interface

CSV Repository

```
public IEnumerable<Person> GetPeople()
{
    var people = new List<Person>();
    if (File.Exists(path))
        using (var sr = new StreamReader(path))
        {
            string l
            while ((
                peop
        }
        }
        return p
    }
```

SQL Repository

```
public IEnumerable<Person> GetPeople()
{
    using (var ctx = new PeopleEntities())
    {
        var people = from p in ctx.DataPersons
                     select new Person...

        return
    }
}
```

Service Repository

```
public IEnumerable<Person> GetPeople()
{
    return serviceProxy.GetPeople();
}
```

No Shared Implementation Code

Interfaces & Abstract Classes in the .NET BCL

- **Abstract Classes with Shared Implementation**
  - MembershipProvider, RoleProvider
  - CollectionBase
- **Interfaces to Add Pieces of Functionality**
  - IDisposable
  - INotifyPropertyChanged, INotifyCollectionChanged
  - IEquatable<T>, IComparable<T>
  - IObservable<T>
  - IQueryable<T>, IEnumerable<T>
- **Base Classes that Implement Interfaces / Inherit from Abstract Classes**
  - SqlMembershipProvider
  - SqlConnection, OdbcConnection, EntityConnection
  - List<T>, ObservableCollection<T>

Updating Interfaces

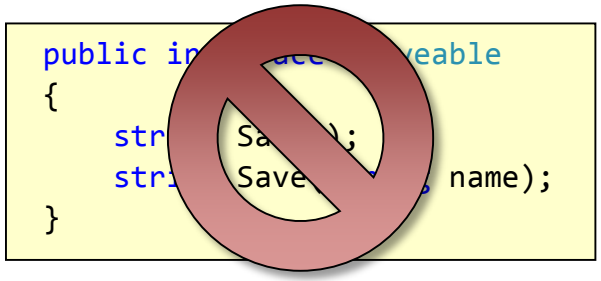
- **Interfaces are a Contract**
  - No Changes after Contract is Signed

- **Adding Members Breaks Implementation**
- **Removing Members Breaks Usage**

- **Inheritance is a Good Way to Add to an Interface**

Adding Members with Inheritance

```
public interface ISaveable
{
    string Save();
}
```



- Breaks Existing Implementers

```
public interface INamedSaveable :
   ISaveable
{
    string Save(string name);
}
```

- Existing ISaveable Still Works

Dependency Injection

- **Loosely Coupled Code**
- **Make "Something Else" Responsible for Dependent Objects**
- **Design Patterns**
  - Constructor Injection
  - Property Injection
  - Method Injection
  - Service Locator
- **Dependency Injection Containers**
  - Unity, StructureMap, Autofac, Ninject, Castle Windsor, and many others

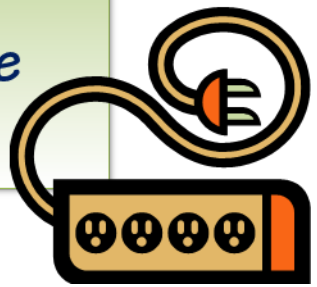
Mocking

- **Create "Placeholder" Objects**
  - In-Memory
  - Only Implement Behavior We Care About
- **Great for Unit Testing**
- **Mocking Frameworks**
  - RhinoMocks
  - Microsoft Fakes
  - Moq

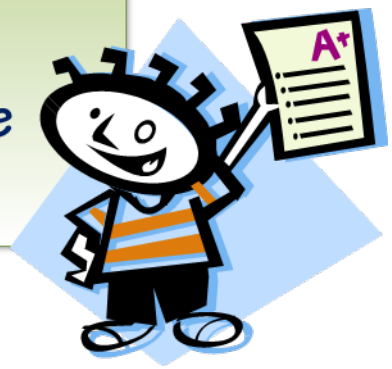
Why Interfaces?

Maintainable

Extensible



Easily Testable



Interfaces help us get there

Goals

- **Learn the "Why"**
  - Maintainability
  - Extensibility
- **Implement Interfaces**
  - .NET Framework Interfaces
  - Custom Interfaces
- **Create Interfaces**
  - Add Abstraction
- **Peek at Advanced Topics**
  - Mocking
  - Unit Testing
  - Dependency Injection

Summary

- **The "What" of Interfaces**

- **Best Practice**

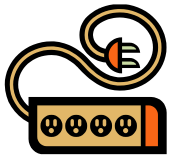
Program to an abstraction rather than a concrete type

or

Program to an interface rather than a concrete class

  - Create Maintainable Code

- **Create & Implement a Custom Interface**
  - Use Abstraction to add Extensibility



Summary

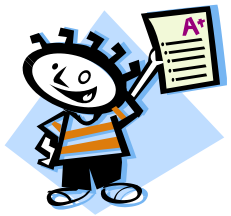
- **Explicit Implementation**

```
string ISaveable.Save()
{
  return "ISaveable Save";
}
```

```
Catalog catalog = new Catalog();
catalog.Save(); // "Catalog Save"

ISaveable saveable = new Catalog();
saveable.Save(); // "ISaveable Save"
```

- **Dynamic Loading & Unit Testing**
  - Fake Repository for Testability



- **Advanced Topics**
  - Interface Segregation Principle
  - Dependency Injection
  - Mocking

C# Interfaces

A Practical Guide to Interfaces

Jeremy Clark
www.jeremybytes.com
jeremy@jeremybytes.com

pluralsight
hardcore developer training