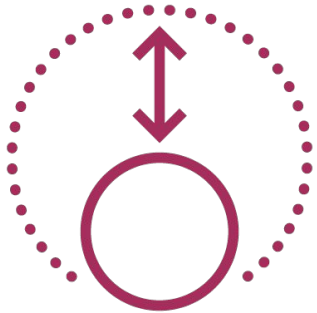
C# Interfaces

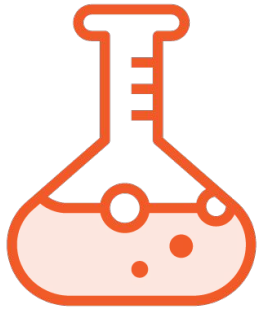
A PRACTICAL GUIDE TO INTERFACES

Why Interfaces?

Maintainable



Extensible



Easily testable

Goals

Learn the “Why”

- Maintainability
- Extensibility

Implement Interfaces

- .NET Framework Interfaces
- Custom Interfaces

Goals

Create Interfaces
- Add Abstraction

Peek at Advanced Topics
- Mocking
- Unit Testing
- Dependency Injection

Pre-requisites

Basic Understanding of C#

- Classes
- Inheritance
- Properties
- Methods

Interfaces, Abstract Classes, and Concrete Classes

What are Interfaces?

Interface

Interfaces describe a group of related functions that can belong to any class or struct.

Microsoft

What are Interfaces?

Contract

Public set of members

- Properties
- Methods
- Events
- Indexers

Regular Polygons

3 or more sides

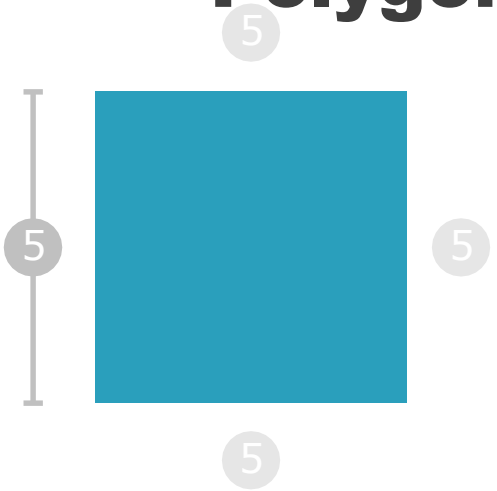
Each side has the same length

Scenario: Regular Polygons

3 or more sides

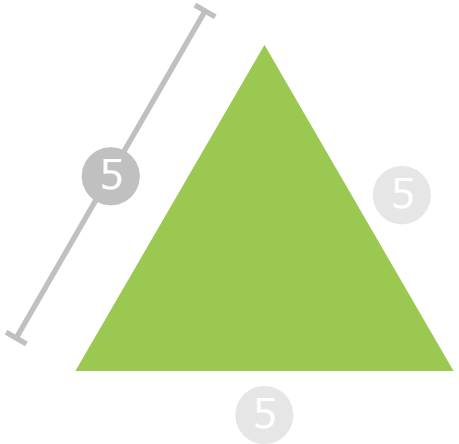
Each side has the same length

Scenario: Regular Polygons



Square

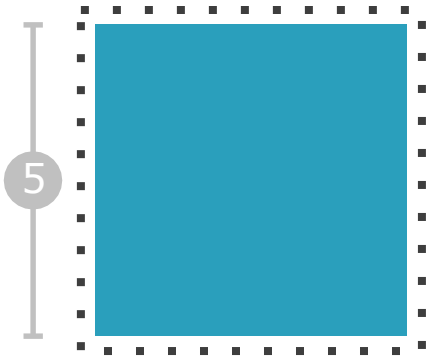
4 sides
Each side has same length



Equilateral Triangle

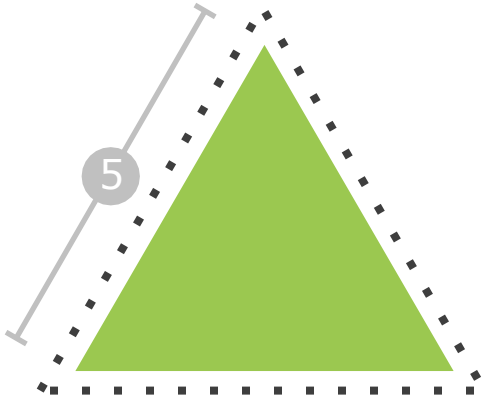
3 sides
Each side has same length

Perimeter



Perimeter =
Number of Sides x Side Length

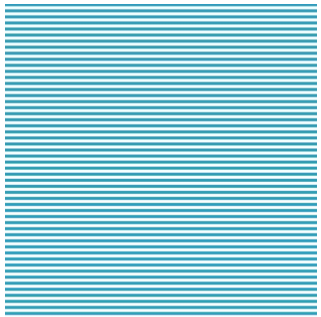
Perimeter = 4 x 5
Perimeter = 20



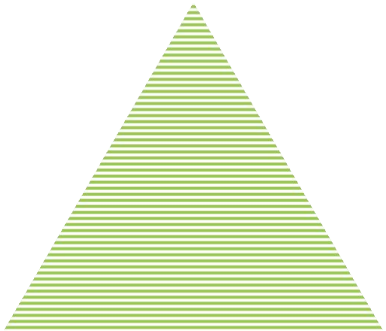
Perimeter =
Number of Sides x Side Length

Perimeter = 3 x 5
Perimeter = 15

Area

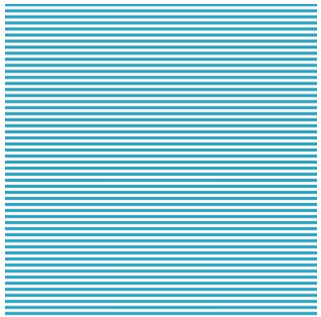


Area =
Side Length x Side Length



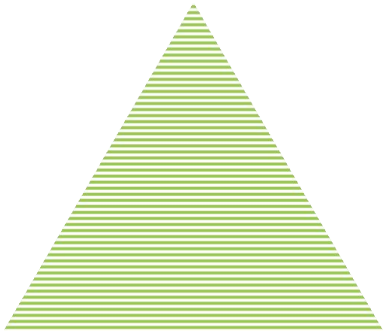
Area =
Side Length x Side Length
x Square Root of 3
Divided by 4

Area



Area = 5 x 5

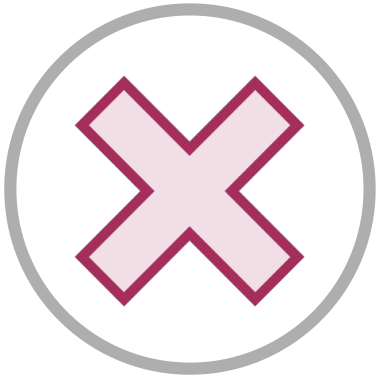
Area = 25



Area = 5 x 5 x Sqrt(3) /4

Area = 10.8 (approximately)

Concrete Class, Abstract Class, or Interface?



Concrete Class

No Compile-time checking

Abstract Class

Compile-time checking

Interface

Compile-time Checking

```
public abstract class
AbstractRegularPolygon
{
    public double GetPerimeter()
    {
        return NumberOfSides * SideLength;
        }

}
```

Comparison: Implementation Code

Abstract Classes may contain implementation

Interfaces may not contain implementation (declarations only)

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

Comparison: Inheritance

Inherit from a single Abstract Class (Single Inheritance)

Implement any number of Interfaces

```
public abstract class
AbstractRegularPolygon
{
    public int NumberOfSides { get; set; }
    public int SideLength { get; set; }
    public double GetPerimeter()…
    public abstract double GetArea();
}
```

Comparison: Access Modifiers

Abstract Classes Members can have access modifiers

```
public interface IRegularPolygon
{
    int NumberOfSides { get; set; }
    int SideLength { get; set; }
    double GetPerimeter();
    double GetArea();
}
```

Comparison: Access Modifiers

Interface Members are automatically public

Comparison: Valid Members

Abstract Classes	Interfaces
Fields	Properties
Properties	Methods
Constructors	Events
Destructors	Indexers
Methods	
Events	
Indexers	

Comparison Summary

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

Comparison Summary

Abstract Classes	Interfaces
May contain implementation code	May not contain implementation code
A class may inherit from a single base class	A class may implement any number of interfaces
Members have access modifiers	Members are automatically public
May contain fields, properties, constructors, destructors, methods, events and indexers	May only contain properties, methods, events, and indexers

Summary

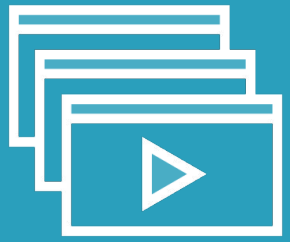
The “What” of Interfaces

Public set of members:

- Properties
- Methods
- Events
- Indexers

Compiler-enforced Implementation

Comparison between Abstract Classes and Interfaces



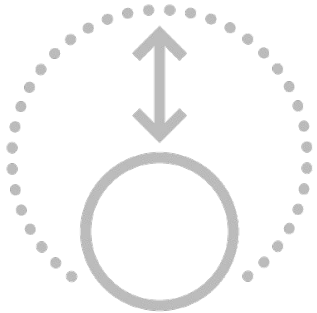
UP NEXT:

The "Why" of Interfaces

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>
- Stack / Stack<T>
- Dictionary<TKey, TValue>
- ObservableCollection<T>
- + Custom Types

FIF

LIF

```
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

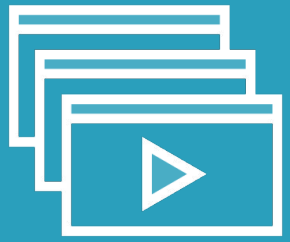
Summary

Concrete Class

- Brittle / Easily Broken

Interface

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



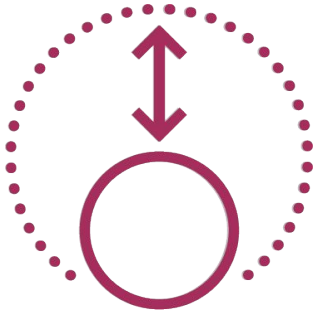
UP NEXT:

The “How” of Interfaces

Creating Interfaces to Add Extensibility

Why Interfaces?

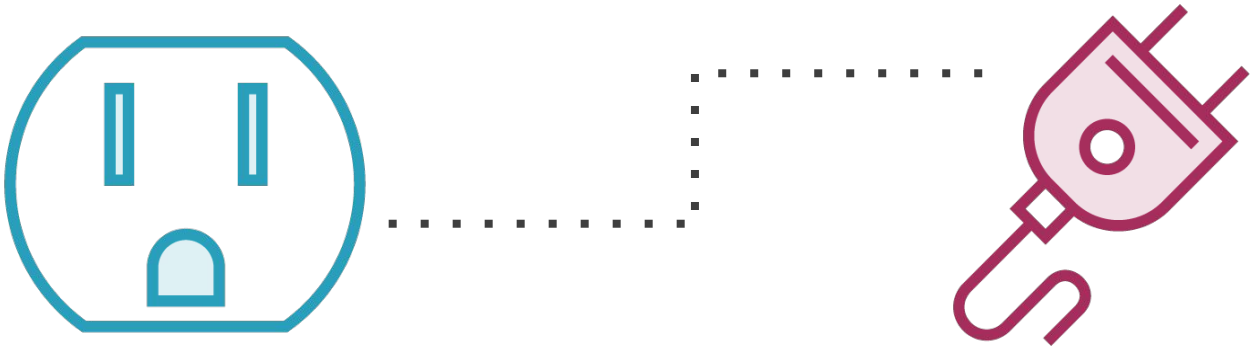
Maintainable



Extensible

Easily testable

Extensible



Different Data Sources

Relational Databases
- Microsoft SQL Server, Oracle, MySQL, etc.

Document / Object Databases (NoSQL)
- MongoDB, Hadoop, RavenDB, etc.

Text Files
- CSV, XML, JSON, etc.

SOAP Services
- WCF, ASMX Web Service, Apache CXF, etc.

REST Services
- WebAPI, WCF, Apache CXF, JAX-RS, etc.

Cloud Storage
- Microsoft Azure, Amazon AWS, Google Cloud SQL

Repository Pattern

Mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects.

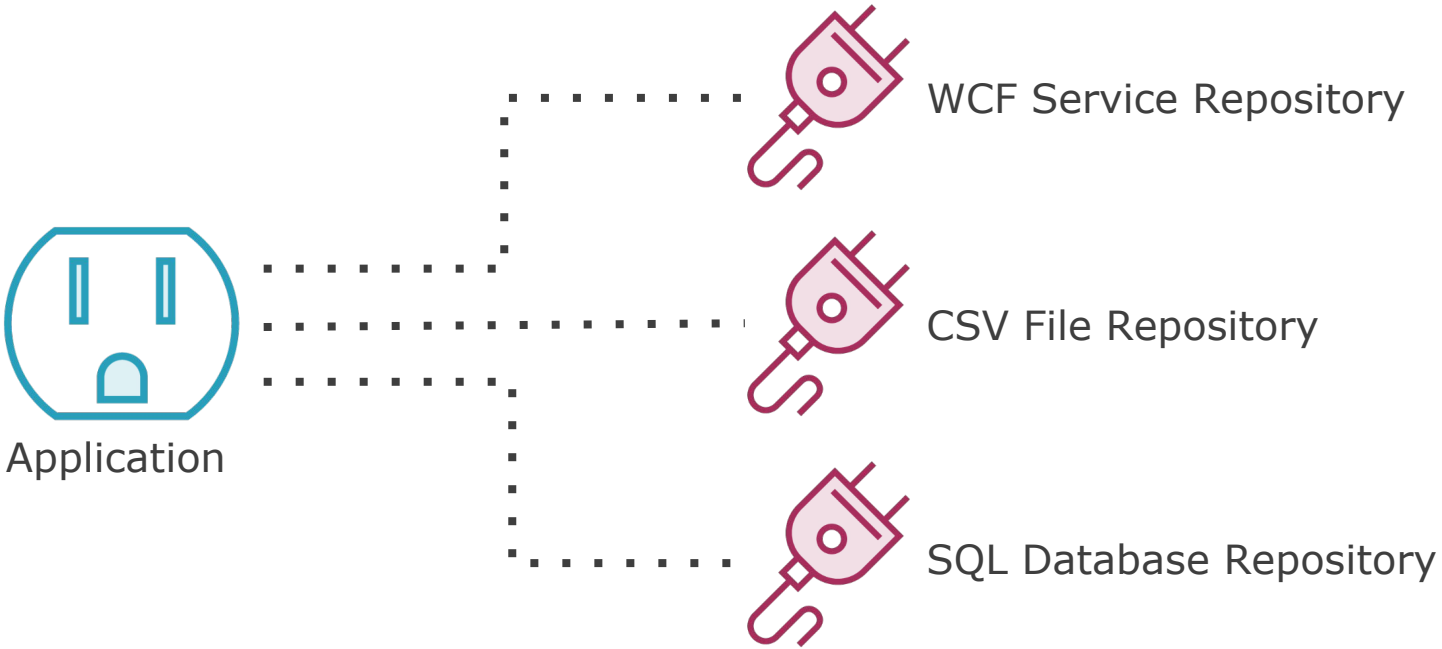
– Fowler, et al. Patterns of Enterprise Application Architecture. Addison-Wesley, 2003

Repository Pattern

Layer to separate our application from the data storage technology

- Application
- Repository
- Data Storage

Pluggable Repositories



Simple Repository

Data Access Operations

- C Create
- R Read
- U Update
- D Delete

Creating a Repository Interface

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

    IEnumerable<Person> GetPeople();

    Person GetPerson(string lastName);

    void UpdatePerson(string lastName,  Person
          updatedPerson);

    void UpdatePeople(IEnumerable<Person>
          updatedPeople);

    void DeletePerson(string lastName);
}
```

C

R

U

D

Summary

Repository Pattern

- Create
- Read
- Update
- Delete

How to Create and Implement a Custom Interface

- IPerson Repository

Easy Extensibility

UP NEXT:

Explicit Interface Implementation

Explicit Interface Implementation

Explicit Implementation

Implement Interface

Explicitly Implement Interface

Class with No Interface

Declaration

```
public class Catalog :
ISaveable
{
  public string Save()
  {
    return "Catalog Save";
  }

  // Other members not shown
}
```

Usage

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

Standard Interface Implementation

Declaration

```
public interface ISaveable
{
  string Save();
}
public class Catalog :
ISaveable
  {
    public string Save()
    {
     return "Catalog Save";
    }

    // Other members not shown
  }
```

Usage

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

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

Explicit Interface Implementation

Declaration

```
public class Catalog :
ISaveable
{
   public string Save()
   {
    return "Catalog Save";
   }
   string ISaveable.Save()
   {
     return "ISaveable Save";
   }
   // Other members not shown
}
```

Concrete Type

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

Interface Variable

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

Cast to Interface

```
((ISaveable)catalog).Save();
//  "ISaveable Save"
```

Explicit Interface Implementation

Declaration

```
public class Catalog :
ISaveable
{
   string ISaveable.Save()
   {
     return "ISaveable Save";
    }
   // Save() deleted
   // Other members not shown
 }
```

Concrete Type

```
Catalog catalog = new Catalog();
catalog.Save(); // **COMPILER
ERROR**
```

Interface Variable

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

Cast to Interface

```
((ISaveable)catalog).Save();
//  "ISaveable Save"
```

Mandatory Explicit Implementation

Declaration A

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

Declaration B

```
public interface
IVoidSaveable
{
  void Save();
}
```

Implementation

```
public class Catalog :
  ISaveable, IVoidSaveable
{
  public string Save()
  {
    return "Catalog Save";
  }
   void
   IVoidSaveable.Save()

  {
    // no return value
  }
   // Other members not
   shown
```

Mandatory Explicit Implementation

Declaration A

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

Declaration B

```
public interface
IVoidSaveable
{
  void Save();
}
```

Implementation

```
public class Catalog :
  ISaveable, IVoidSaveable
{
  string ISaveable.Save()
  {
    return "ISaveable
    Save";
  }
  public void Save()

  {
    // no return value
  }
     // Other members not
           shown
```

Mandatory Explicit Implementation

Declaration A

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

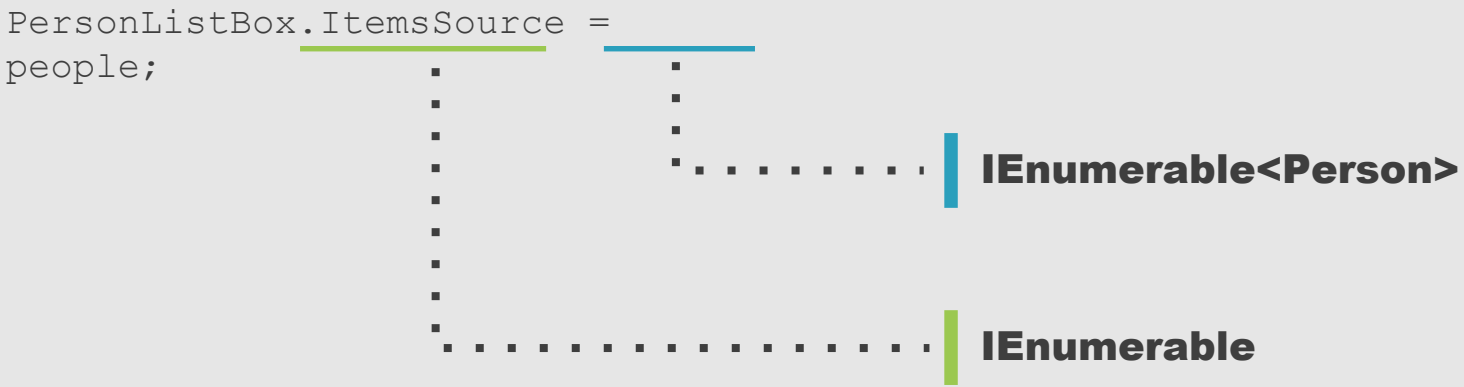
Declaration B

```
public interface
IVoidSaveable
{
  void Save();
}
```

Implementation

```
public class Catalog :
  ISaveable, IVoidSaveable
{
  string ISaveable.Save()
  {
    return "ISaveable
    Save";
  }
  void IVoidSaveable.Save()

  {
    // no return value
  }
     // Other members not
           shown
```

Type Mismatch?



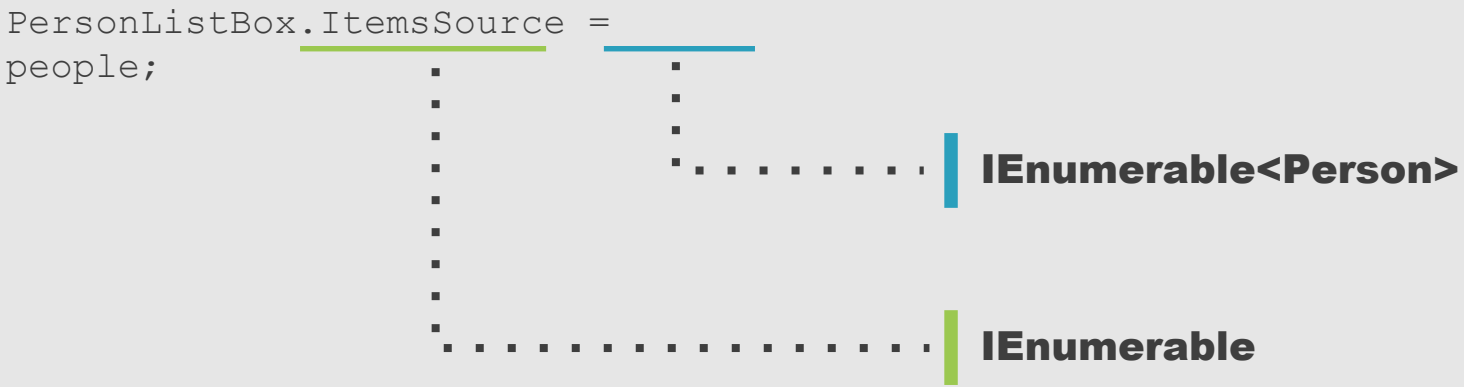
```
public interface IEnumerable<T> :
IEnumerable
```

Interface Inheritance

IEnumerable<T> inherits IEnumerable

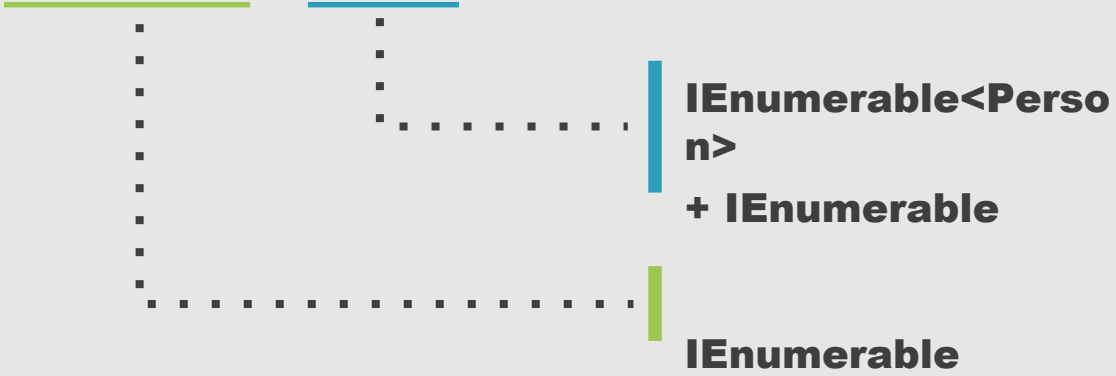
When a class implements IEnumerable<T>, it must also implement IEnumerable

Type Mismatch?



Type Mismatch?

```
PersonListBox.ItemsSource =
people;
```



Interface Members

IEnumerable<T> Members

```
public interface IEnumerable<T>:
IEnumerable
  {
    IEnumerator<T> GetEnumerator();
  }
```

IEnumerable Members

```
public interface IEnumerable
  {
    IEnumerator
    GetEnumerator();
  }
```

Summary

Standard Implementation

Explicit Implementation

- **Save method for class**
- **Save method for interface**

Mandatory Explicit Implementation

- **Methods with Different Return Types**

Interface Inheritance

- **IEnumerable<T> and IEnumerable**

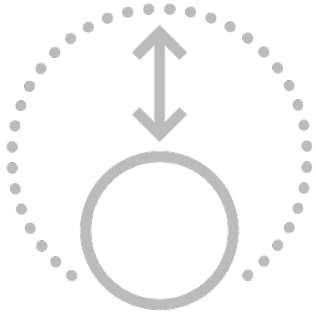
UP NEXT:

Interfaces and Dynamic Loading

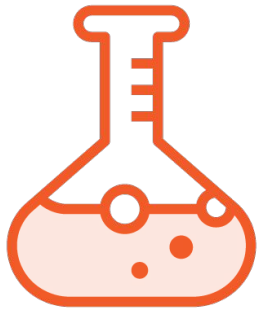
Interfaces and Dynamic Loading

Why Interfaces?

Maintainable



Extensible



Easily testable

Best Practice

Program to an abstraction rather than a concrete type

Program to an interface rather than a concrete class

Program to an Interface

```
private void FetchData(string repositoryType)
{
    ClearListBox();
    IPersonRepository repository =
        RepositoryFactory.GetRepository(repositoryType);
    var people = repository.GetPeople();  foreach (var
    person in people)
        PersonListBox.Items.Add(person);
    ShowRepositoryType(repository);
}
```

No Reference to Concrete Types

Compile-Time Factory

```
public static class RepositoryFactory
{
    public static IPersonRepository GetRepository(  string
      repositoryType)
    {
      IPersonRepository repo = null;  switch (repositoryType)
      {
          case "Service":  repo = new ServiceRepository ();  break;
          case "CSV":  repo = new CSVRepository ();
              break;
          case "SQL":  repo = new SQLRepository ();  break;
          default:
              throw new ArgumentException ("Invalid Repository Type" );
      }
      return repo;
   }
}
```

Factory Comparison

Compile-Time Factory

Has a Parameter

- Caller decides which repository to use

Compile-Time Binding

- Factory needs references to repository assemblies

Dynamic Factory

No Parameter

- Factory returns a repository based on configuration

Run-Time Binding

- Factory has no compile-time references to repository assemblies

Dynamic Loading

Get Type and Assembly from Configuration

Load Assembly through Reflection

Create a Repository Instance with the Activator

Dynamic Loading

```
public static class RepositoryFactory
{
  public static IPersonRepository GetRepository()
  {
    string typeName =
      ConfigurationManager.AppSettings["RepositoryType"];

    Type repoType = Type.GetType(typeName);
    object repoInstance = Activator.CreateInstance(repoType);
    IPersonRepository repo = repoInstance as IPersonRepository;
    return repo;

  }
}
```

Unit Testing

Testing small pieces of code

- Usually on the method level

Testing in isolation

- Eliminate outside interactions that might break the test
- Reduce the number of objects needed to run the test

Note: We still need Integration Testing

- Testing that the pieces all work together

What We Want to Test

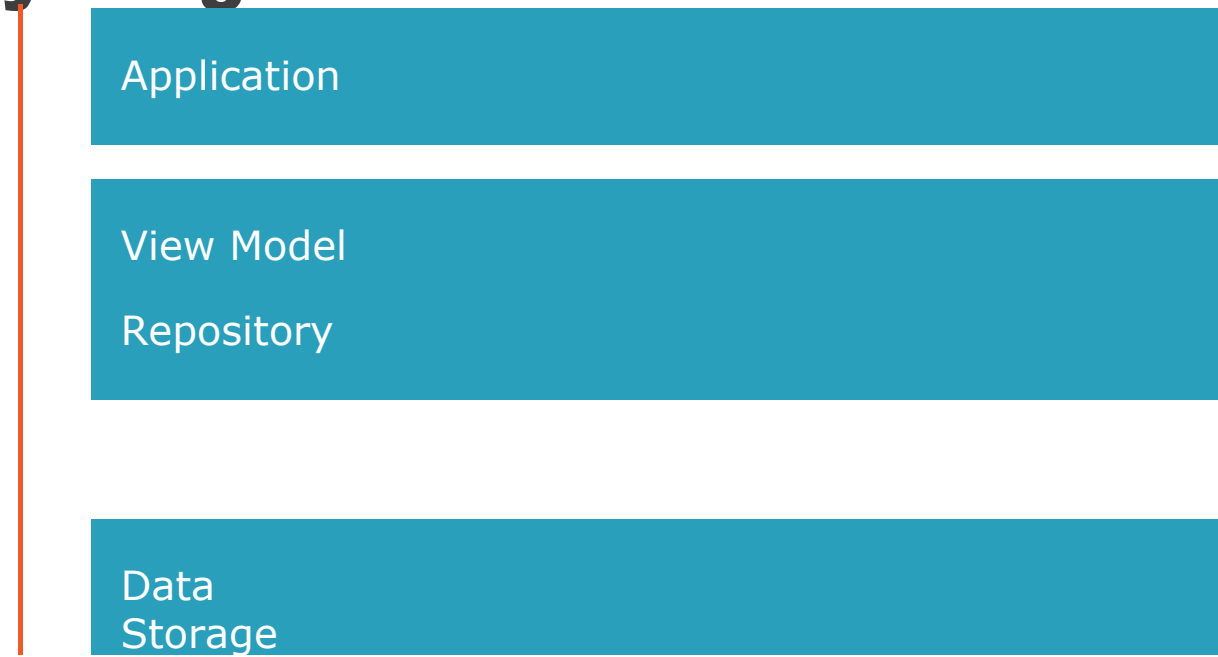
```
public partial class MainWindow : Window
{
    private void FetchButton_Click(object sender, RoutedEventArgs
    e)
    {
      ClearListBox();
      IPersonRepository repository =

      RepositoryFactory.GetRepository();   var people =

      repository.GetPeople();

      foreach (var person in people)
        PersonListBox.Items.Add(person);

      ShowRepositoryType(repository);
    }
```

What We Want to Test

```
public partial class MainWindow : Window
{
   private void FetchButton_Click(object sender, RoutedEventArgs
   e)
   {
     ClearListBox();
     IPersonRepository repository =

     RepositoryFactory.GetRepository();   var people =

     repository.GetPeople();

     foreach (var person in people)
       PersonListBox.Items.Add(person);

     ShowRepositoryType(repository);
   }
```

Additional Layering

Very Simple MVVM Implementation



Isolating Code

Move Functionality to a View Model

- Eliminates dependency on UI objects

Add a Fake Repository

- Eliminates dependency on network, file system, or SQL database
- Ensures consistent behavior

Remember: Not testing Repository here.

Testing “Fetch Data” functionality in application code.

Summary

Program to an Interface only
Dynamic Loading / Late Binding

Unit Testing

- Application Layering
- Fake Repository

UP NEXT:

Where to go Next

Advanced Interface Topics

WHERE TO GO NEXT

Jeremy Clark

DEVELOPER BETTERER

@jeremybytes

www.jeremybytes.com

Overview

Best Practices

- Interface Segregation Principle
- Choosing Between Abstract Class and Interface
- Updating Interfaces

Overview

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

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

`IEnumerable<T>`

If We Need to

- Iterate over a Collection / Sequence
- Data Bind to a List Control
- Use LINQ functions

Granular Interfaces

```
ICollection<T>
```

If We Need to

- Add/Remove Items in a Collection
- Count Items in a Collection
- Clear a Collection

Granular Interfaces

`IList<T>`

If We Need to

- Control the Order Items in a Collection
- Get an Item by the Index

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

`IEnumerable<T>`

If We Need to

- Iterate over a Collection / Sequence
- Data Bind to a List Control

`ICollection<T>`

If We Need to

- Add/Remove Items in a Collection
- Count Items in a Collection
- Clear a Collection

`IList<T>`

If We Need to

- Control the Order Items in a Collection
- Get an Item by the Index

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);
}
```

Better Segregation

```
public interface IPersonRepository : IReadOnlyPersonRepository
{
    void AddPerson(Person newPerson);
    void UpdatePerson(string lastName, Person updatedPerson);

    void DeletePerson(string lastName);


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

Comparison Summary

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

```
public abstract class AbstractRegularPolygon
{
    public int NumberOfSides { get; set; }  public int
    SideLength { get; set; }

    public AbstractRegularPolygon(int sides, int length)
    {
        NumberOfSides = sides;  SideLength = length;
    }

    public double GetPerimeter()
    {
        return NumberOfSides * SideLength;
    }

    public abstract double GetArea();
}
```

Abstract Class

Lots of Shared Code

Person Repository

CSV Repository

```
public IEnumerable<Person> GetPeople()
{
    var people = new List<Person>();  if
    (File.Exists(path))
        using (var sr = new StreamReader(path))
        {
          string line;
          while ((line = sr.ReadLine()) != null)...
              people.Add(per);
        }
             }
             return people;
        }
```

Person Repository

SQL Repository

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

        return people.ToList();
    }
}
```

Person Repository

Service Repository

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

Interface

No Shared Implementation Code

Interfaces & Abstract Classes in the .NET BCL

Abstract Classes with Shared Implementation

MembershipProvider, RoleProvider

CollectionBase

Interfaces & Abstract Classes in the .NET BCL

Interfaces to Add Pieces of Functionality

IDisposable

INotifyPropertyChanged,
INotifyCollectionChanged

IEquatable<T>,

IComparable<T>

IObservable<T>

IQueryable<T>,
IEnumerable<T>

Interfaces & Abstract Classes in the .NET BCL

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();
}
```

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

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

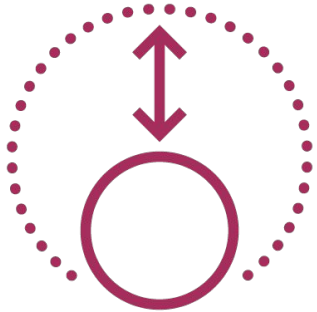
Great for Unit Testing Mocking

Frameworks

- RhinoMocks
- Microsoft Fakes
- Moq

Why Interfaces?

Maintainable



Extensible

Easily testable

Goals

Learn the ‘Why”

- Maintainability
- Extensibility

Implement Interfaces

- .NET Framework Interfaces
- Custom Interfaces

Goals

Create Interfaces
- Add Abstraction

Peek at Advanced Topics
- Mocking
- Unit Testing
- Dependency Injection

Summary

The “What” of Interfaces

Summary

Best Practice

Program to an abstraction rather than a concrete type

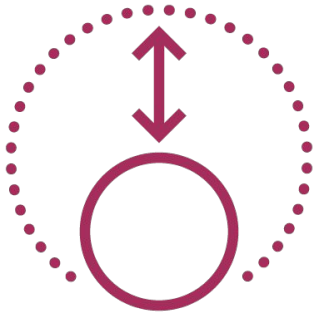
Summary

Program to an interface rather than a concrete class

Summary

Create Maintainable Code

Summary



Create & Implement a Custom Interface

- Use Abstraction to add Extensibility

Summary

Dynamic Loading & Unit Testing

- Fake Repository for Testability

Summary

Advanced Topics

Interface Segregation Principle
Dependency Injection

Mocking

Further Courses:

- Dependency Injection
- Solid Design Principles
- Model View / View Model Pattern
- Unit Testing
- Test-Driven Development
- Mocking