.NET for C Programmers

Day 1: Core C#

- Part 1
  - .NET and Visual Studio
  - Flow control in C#
  - Types and assemblies
  - Classes and objects
  - Interfaces
  - Change notification
- Part 2
  - Collections
  - Attributes
  - Events
  - Generics
  - Iteration

Core C#

.NET and Visual Studio

Logistics

- Retrieve slides and code from GitHub
- Survey of knowledge on the content

Core C#

.NET and Visual Studio

Overview: .NET and VS.NET

- What is .NET?
- Languages for .NET
- Visual Studio.NET
- Demo

What is .NET?

- Object oriented programming environment
- Common Language Runtime (CLR)
- Code execution environment (JIT based)
- Set of base classes facilitating development

Languages on .NET

- From Microsoft
  - C#
    - Most popular
    - Current version is 5.0
    - Very robust
  - VB.NET
  - F#
  - Managed C++

Visual Studio .NET

- Integrated development environment
- Focuses on .NET development
- But also can do other platforms such as Node.JS
- Highly extensible

Demo: Visual Studio Overview

- Overview of project types
- Creation of a console application

Core C#

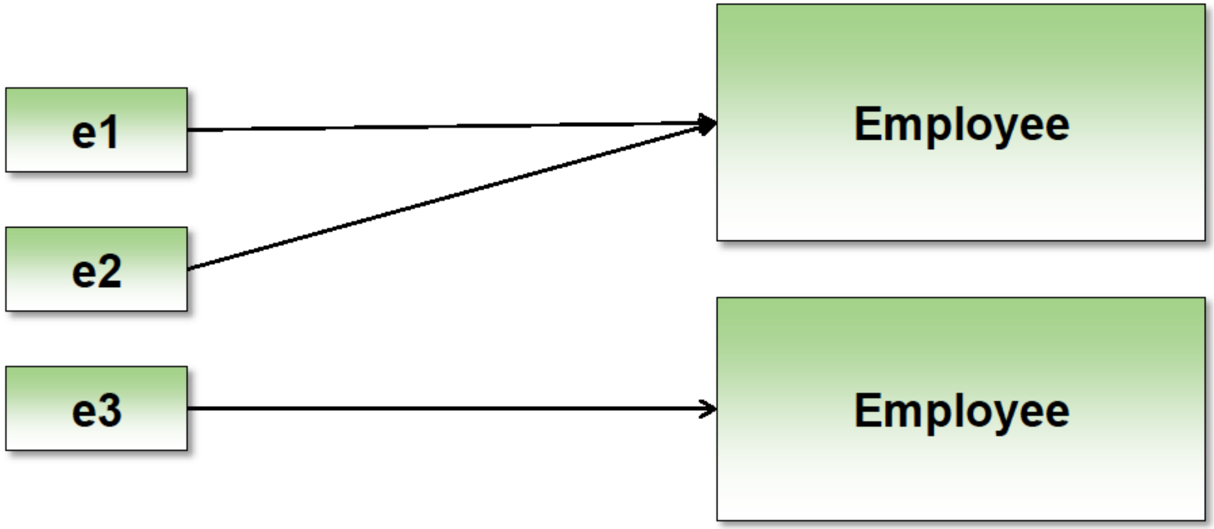
Types and Assemblies

Overview: Types and Assemblies

- Value types
- Enumerations
- Structs
- Interfaces
- Arrays
- Assemblies
- Assembly references
- Garbage collection

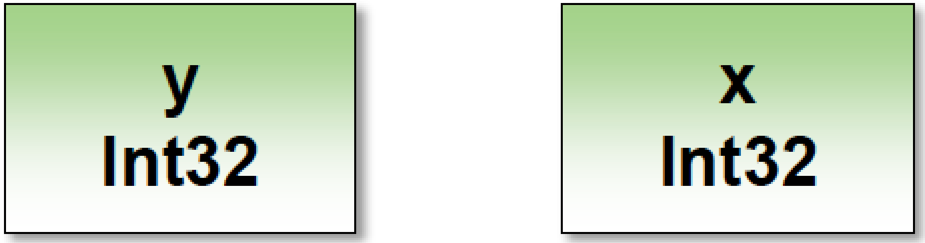
Reference Types

- Variables store a reference to an object
  - Multiple variables can point to the same object
  - Single variable can put to multiple objects over the apps lifetime
  - Objects are allocated on the heap by the new operator



Value Types

- Variables hold the value
  - No pointers or references
  - No object allocated on the heap –lightweight
  - Should be immutable
- **Many built-in primitives are value types**
  - Int32, DateTime, Double

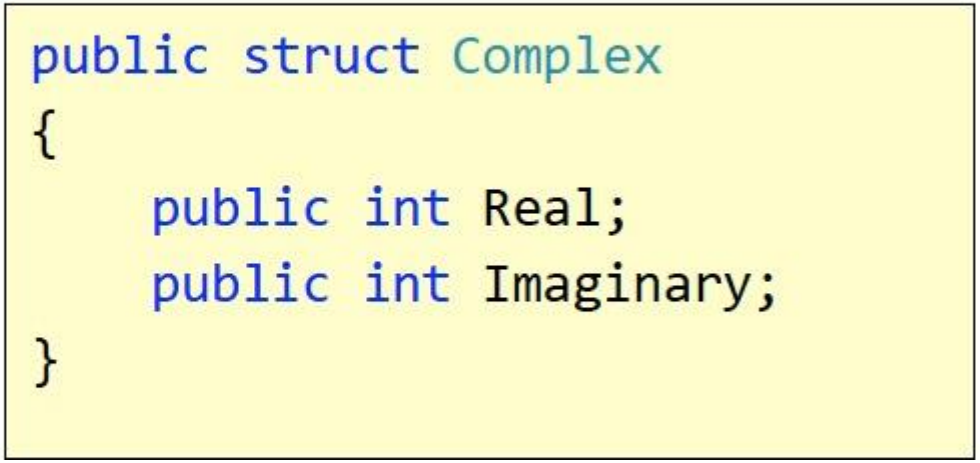


y
Int32

x
Int32

Creating Value Types

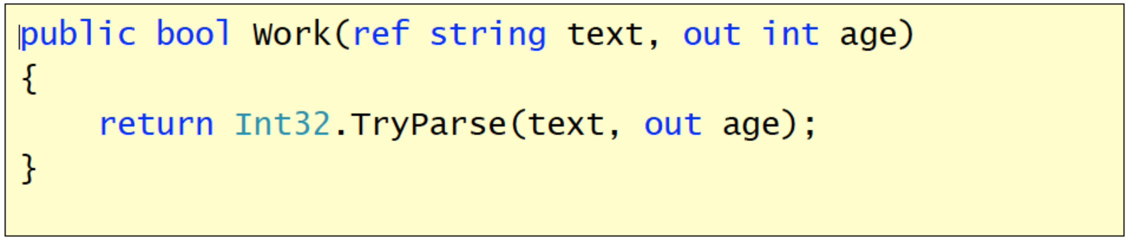
- **Struct definitions create value types**
  - Cannot inherit from a struct (implicitly sealed)
  - Rule of thumb: should be less than 16 bytes



```
public struct Complex
{
    public int Real;
    public int Imaginary;
}
```

Method/Function Parameters

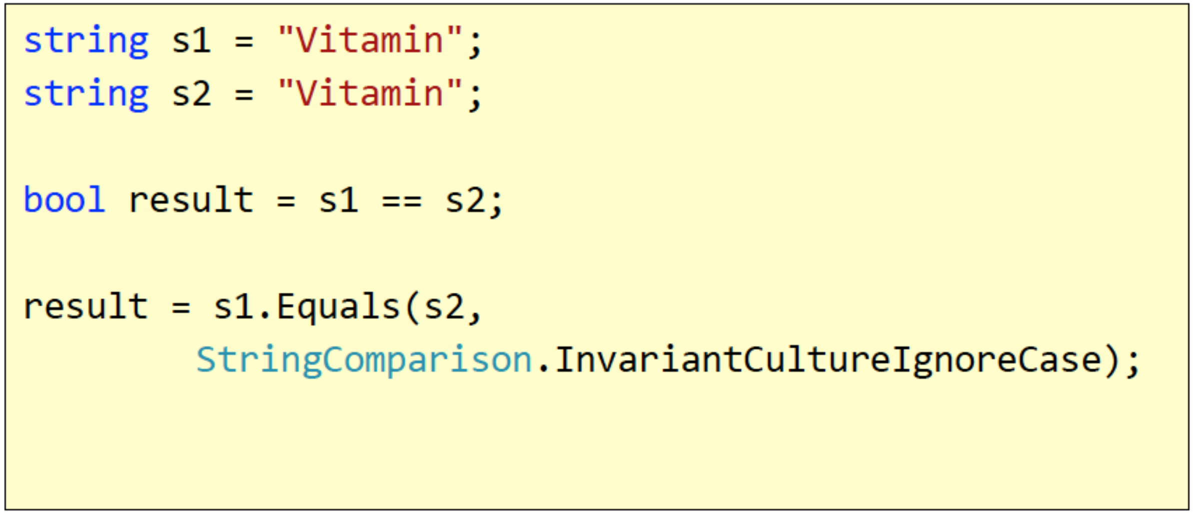
- **Parameters pass “by value”**
  - Reference types pass a copy of the reference
  - Value types pass a copy of the value
  - Changes to value don’t propagate to caller
- **Parameter keywords**
  - ref and out keywords allow pass “by reference”
  - ref parameters requires initialized variable



```
public bool Work(ref string text, out int age)
{
    return Int32.TryParse(text, out age);
}
```

The String Type

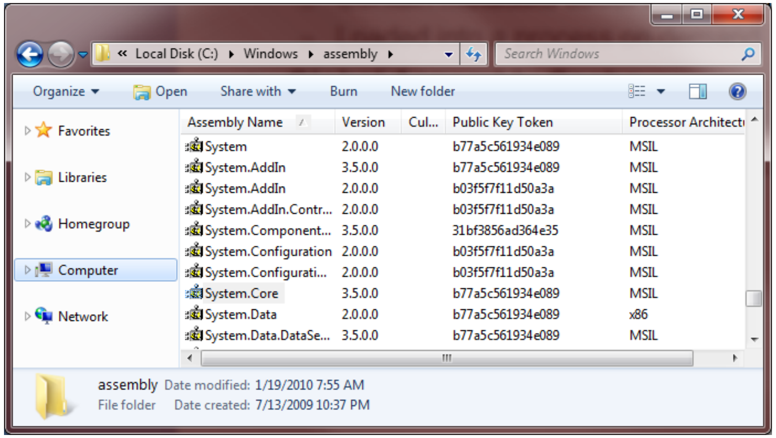
- **Strings are reference types**
  - But behave like value types
  - Immutable
  - Checking for equality performs a string comparison



```
string s1 = "Vitamin";
string s2 = "Vitamin";

bool result = s1 == s2;

result = s1.Equals(s2,
        StringComparison.InvariantCultureIgnoreCase);
```

Assemblies

- **Fundamental building blocks**
  - Implemented as .exe or .dll Files
  - Contain metadata about version and all types inside
- **Global Assembly Cache**
  - A central location to store assemblies for a machine
  - Assembly in the GAC requires a strong name

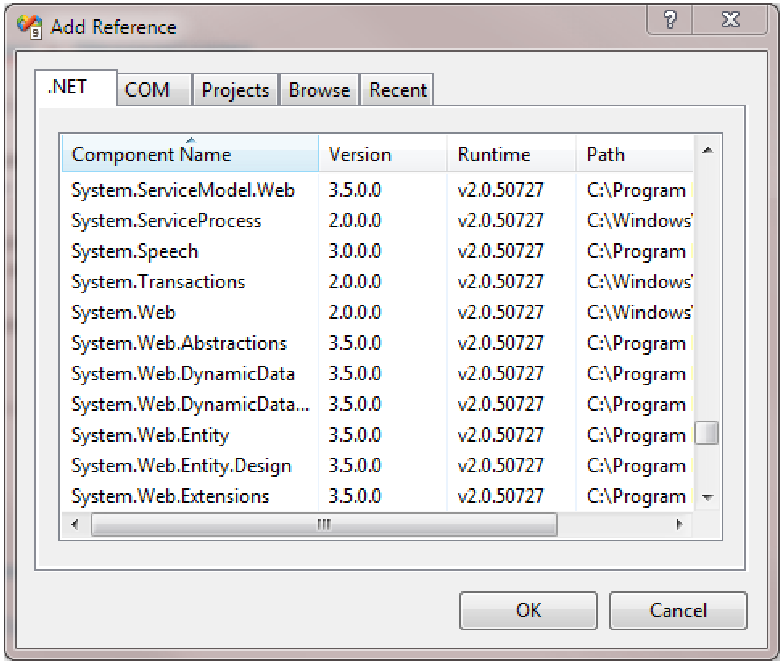


Assembly Name	Version	Cul...	Public Key Token	Processor Architect
System	2.0.0.0		b77a5c561934e089	MSIL
System.AddIn	3.5.0.0		b77a5c561934e089	MSIL
System.AddIn	2.0.0.0		b03f5f7f11d50a3a	MSIL
System.AddIn.Contr...	2.0.0.0		b03f5f7f11d50a3a	MSIL
System.Component...	3.5.0.0		31bf3856ad364e35	MSIL
System.Configuration	2.0.0.0		b03f5f7f11d50a3a	MSIL
System.Configurati...	2.0.0.0		b03f5f7f11d50a3a	MSIL
System.Core	3.5.0.0		b77a5c561934e089	MSIL
System.Data	2.0.0.0		b77a5c561934e089	x86
System.Data.DataSe...	3.5.0.0		b77a5c561934e089	MSIL

assembly Date modified: 1/19/2010 7:55 AM
File folder Date created: 7/13/2009 10:37 PM

References

- **Must load assembly into a process before using types inside**
  - Easy approach –reference the assembly in Visual Studio
  - Assemblies loaded on demand at runtime



Component Name	Version	Runtime	Path
System.ServiceModel.Web	3.5.0.0	v2.0.50727	C:\Program
System.ServiceProcess	2.0.0.0	v2.0.50727	C:\Windows
System.Speech	3.0.0.0	v2.0.50727	C:\Program
System.Transactions	2.0.0.0	v2.0.50727	C:\Windows
System.Web	2.0.0.0	v2.0.50727	C:\Windows
System.Web.Abstractions	3.5.0.0	v2.0.50727	C:\Program
System.Web.DynamicData	3.5.0.0	v2.0.50727	C:\Program
System.Web.DynamicData...	3.5.0.0	v2.0.50727	C:\Program
System.Web.Entity	3.5.0.0	v2.0.50727	C:\Program
System.Web.Entity.Design	3.5.0.0	v2.0.50727	C:\Program
System.Web.Extensions	3.5.0.0	v2.0.50727	C:\Program

Garbage Collection

- **Garbage collector cleans up unused memory**
    - Visits global variables and local variables to determine what is in use
- Generative model
- Self-tuning
- More info at: https://msdn.microsoft.com/en-us/library/ee787088(v=vs.110).aspx

Summary

- **Every type is a value type or reference type**
  - Use struct to create a value type
  - Use class to create a reference type
- **Arrays and strings are reference types**
  - Strings behave like a value type

Demo / Exercise

- Create a console application
- Within a loop
  - Read input from the console
  - Echo it to the console

.NET for C Programmers

Classes and objects

Overview

- Classes vs objects
- Constructors
- Properties
- Fields
- Methods
- Inheritance
- Access modifiers
- Abstract, static, sealed and partial classes

Note

- Here we will examine the parts of C# that help us implement classes
- We will focus tomorrow on using classes to perform OOP

Classes and objects

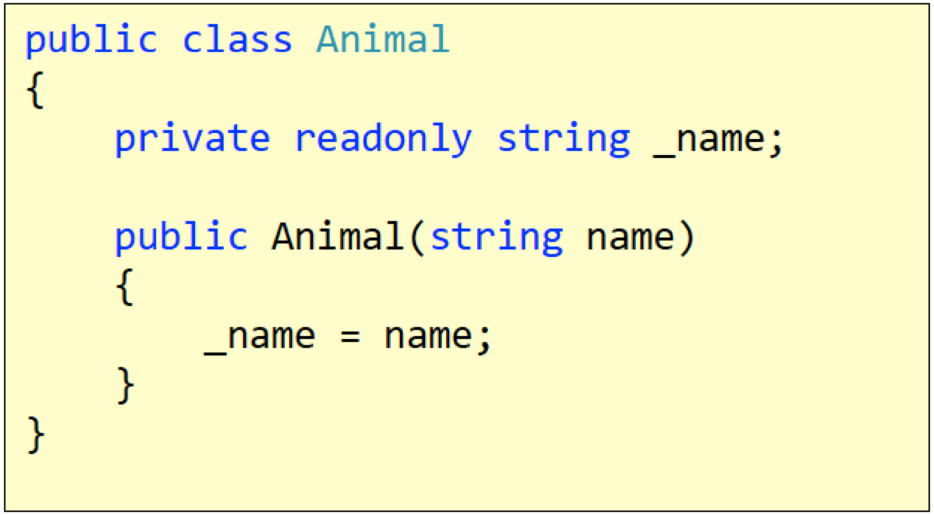
- **Classes define types, which include**
  - State (a set of variables)
  - Behavior (methods)
  - Access (public, private, protected, internal)
- **Objects are instances of a class**
  - Each object is backed by storage in the heap for the defined fields
  - You can create multiple instances of a class
  - Each instance holds different state
  - Each instance has same behavior

Classes in .NET

- Consist of
  - Fields
  - Properties
  - Methods
- Can inherit from 0 or 1 base classes
- Can implement 0 or more interfaces

Fields

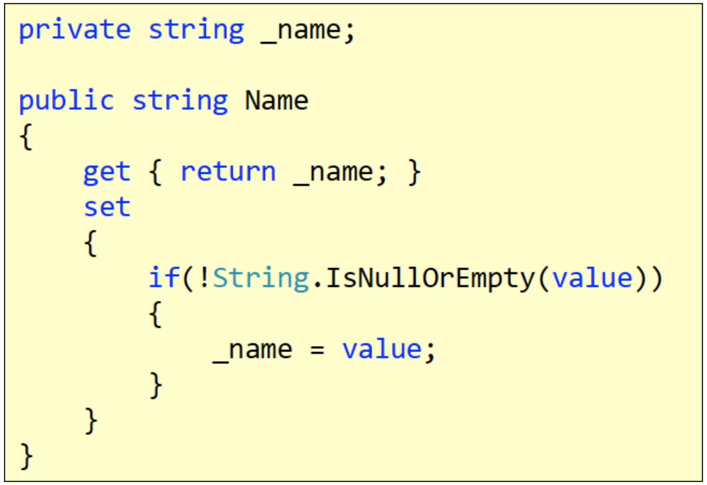
- **Fields are variables of a class**
  - Static fields and instance fields
- **Read-only fields**
  - Can only assign values in the declaration or in a constructor



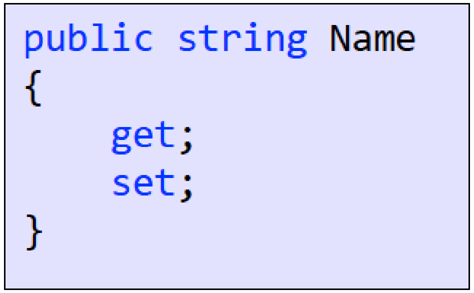
```
public class Animal
{
    private readonly string _name;

    public Animal(string name)
    {
        _name = name;
    }
}
```

Properties

- **Similar to fields, but do not denote a storage location**
  - Every property defines a get and/or a set accessor
  - Often used to expose and control fields
  - Access level for get and set are independent
- **Automatically implemented properties use a hidden field**
  - Only accessible via property



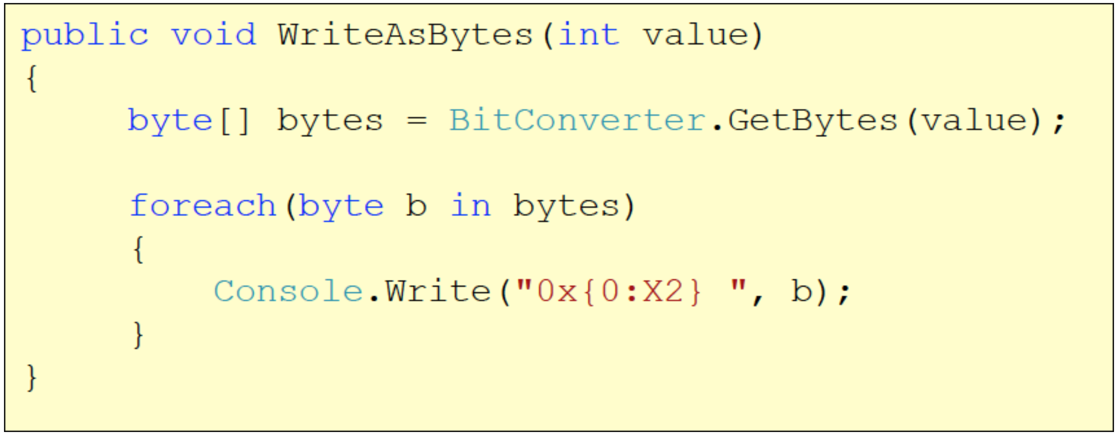
```
private string _name;

public string Name
{
    get { return _name; }
    set
    {
        if(!String.IsNullOrEmpty(value))
        {
            _name = value;
        }
    }
}
```



```
public string Name
{
    get;
    set;
}
```

Methods

- **Methods define behavior**
- **Every method has a return type**
  - void if not value returned
- **Every method has zero or more parameters**
  - Use params keyword to accept a variable number of parameters
- **Every method has a signature**
  - Name of method + parameters (type and modifiers are significant)



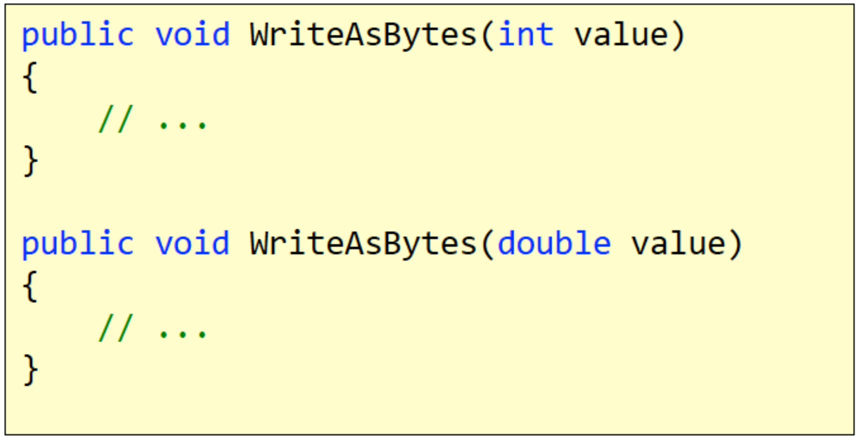
```
public void WriteAsBytes(int value)
{
     byte[] bytes = BitConverter.GetBytes(value);

     foreach(byte b in bytes)
     {
         Console.Write("0x{0:X2} ", b);
     }
}
```

Type of methods

- **Instance methods versus static methods**
  - Instance methods invoked via object, static methods via type
- **Abstract methods**
  - Provide no implementation, implicitly virtual
- **Virtual methods**
  - Can override in a derived class
- **Partial methods**
  - Part of a partial class
- **Extension methods**
  - Described in the LINQ module

Method Overloading

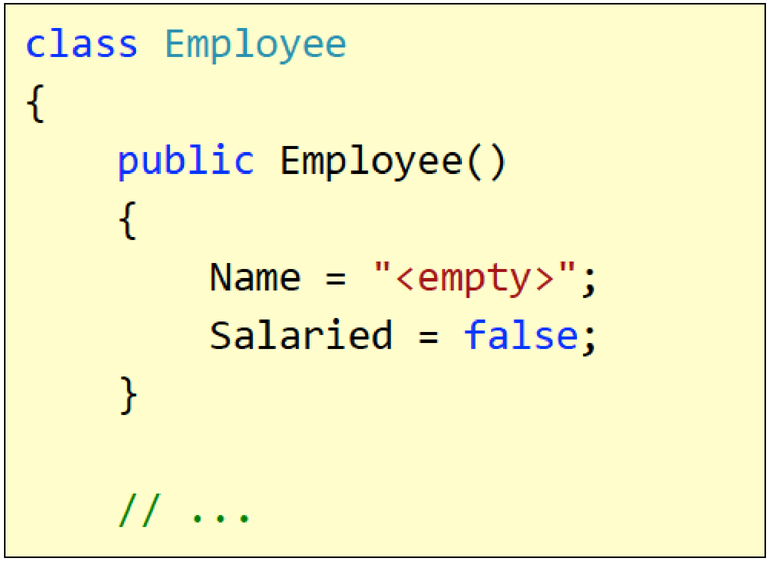
- **Define multiple methods with the same name in a single class**
  - Methods require a unique signature
- **Compiler finds and invokes the best match**
- **Useful for performing a similar task but using different types of parameters**



```
public void WriteAsBytes(int value)
{
    // ...
}

public void WriteAsBytes(double value)
{
    // ...
}
```

Constructors

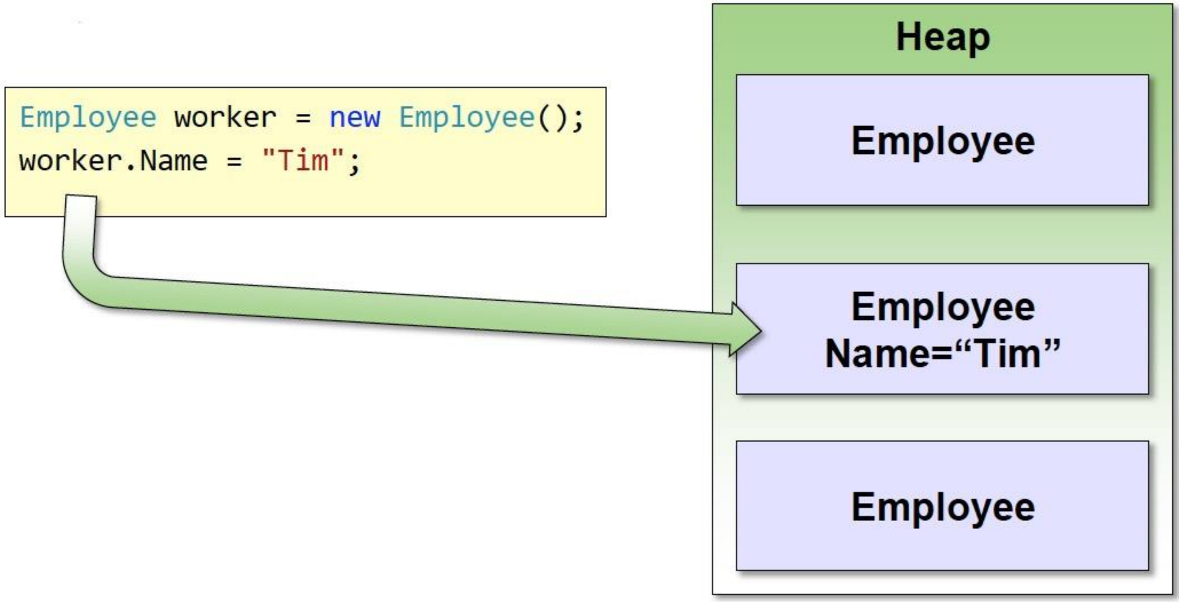
- **Special methods to create objects**
  - Set default values
- **Multiple constructors allowed**
  - Overloaded methods must take different arguments



```
class Employee
{
    public Employee()
    {
        Name = "<empty>";
        Salaried = false;
    }

    // ...
```

Reference Types

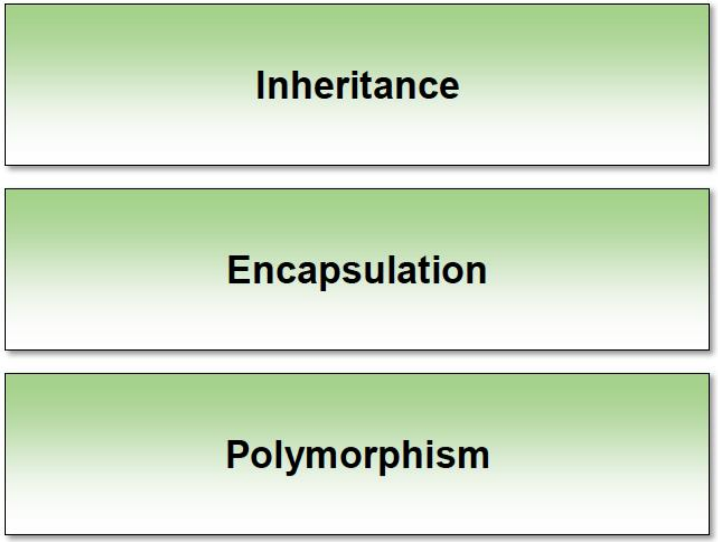
- **Classes create objects which are reference types**
  - Object is stored on the “heap”
  - Variables reference the object instance



```
Employee worker = new Employee();
worker.Name = "Tim";
```

Object Oriented Programming

- C# is an OO language
- We will look at the features in C# for this today, and more on pure OOP tomorrow



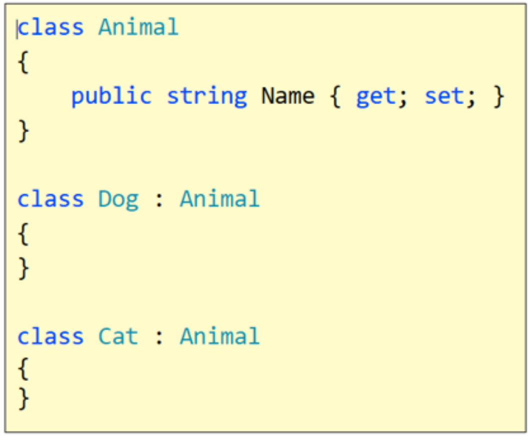
Inheritance

Encapsulation

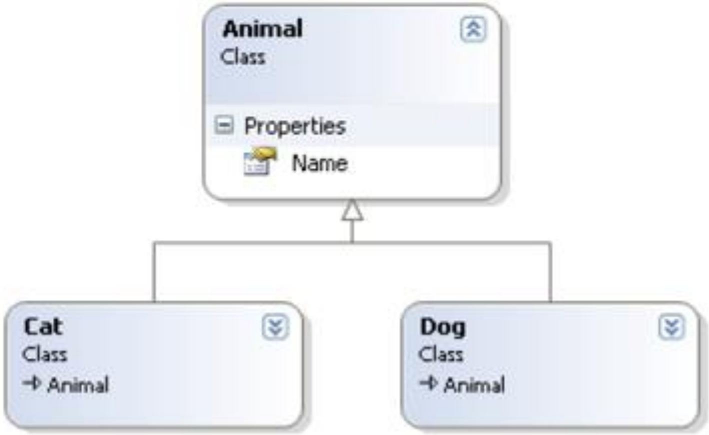
Polymorphism

Inheritance

- **Inheritance is a means of code reuse**
- **Create classes to extend other classes**
  - Classes inherit from System.Object by default
  - Gain all the state and behavior of the base class



```
class Animal
{
    public string Name { get; set; }
}

class Dog : Animal
{
}

class Cat : Animal
{
}
```

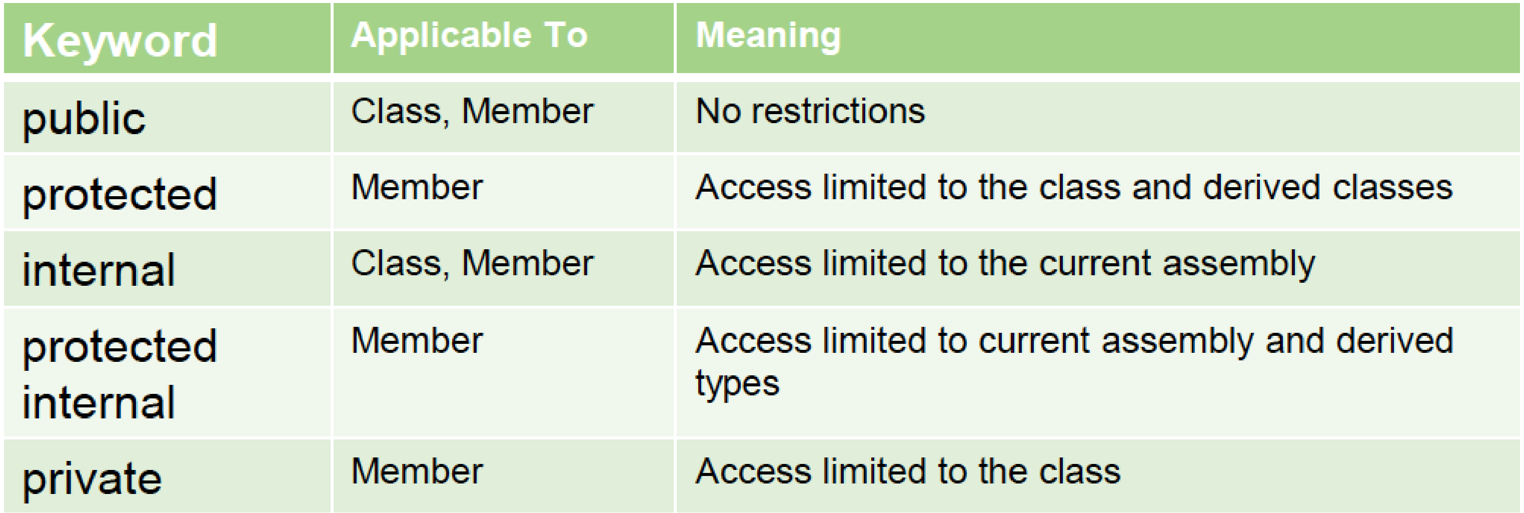


.NET and Inheritance

- A class in .NET can inherit from at most 1 class

Access modifiers

- Control the scope of access to classes, fields, properties and methods
- From derived classes, from other classes in the same or different assemblies



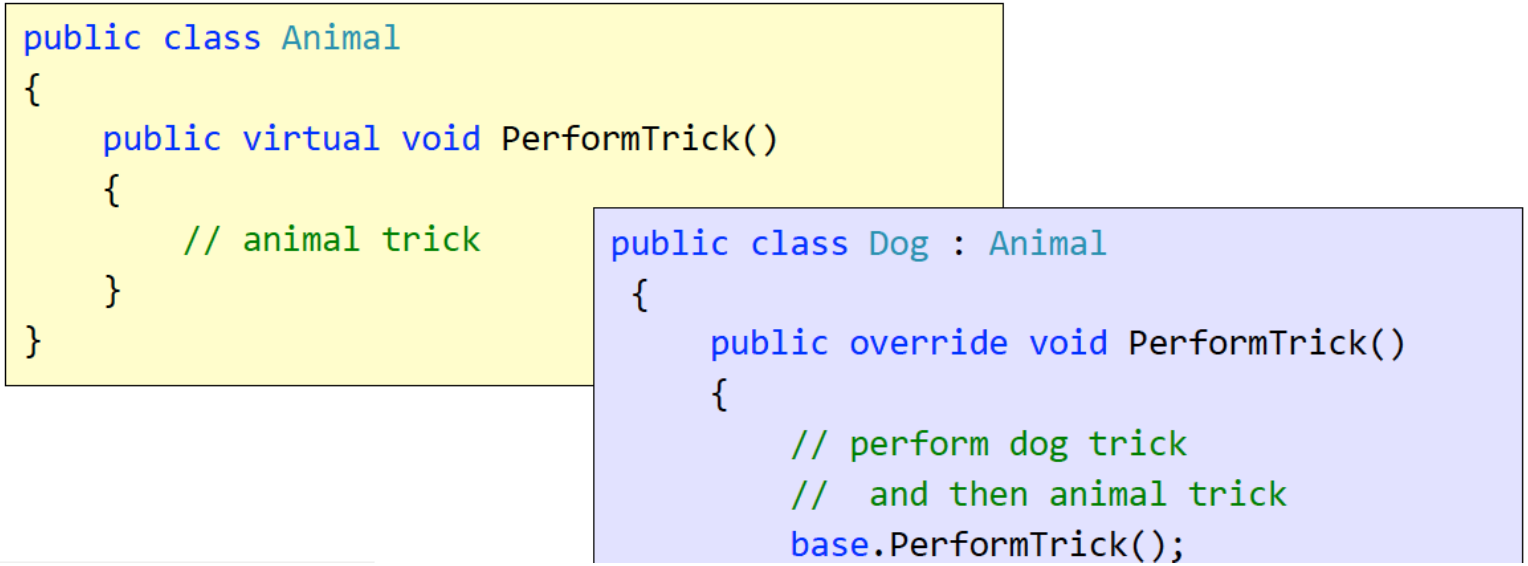
Keyword	Applicable To	Meaning
public	Class, Member	No restrictions
protected	Member	Access limited to the class and derived classes
internal	Class, Member	Access limited to the current assembly
protected internal	Member	Access limited to current assembly and derived types
private	Member	Access limited to the class

Polymorphism

- A means of allowing a derived class to change behavior defined in the base class
- Implemented by declaring a method or property as virtual or abstract
  - Virtual has implementation in that class, and can be overridden in a derived class using **override**
  - Abstract does not have an implementation in the defining class, and all derived classes MUST implement the method (also by override)

Virtual

- **The virtual keyword creates a virtual member**
  - Can override the member in a derived class
  - Members are non-virtual by default
  - Virtual members dispatch on runtime type



```
public class Animal
{
    public virtual void PerformTrick()
    {
        // animal trick
    }
}
```

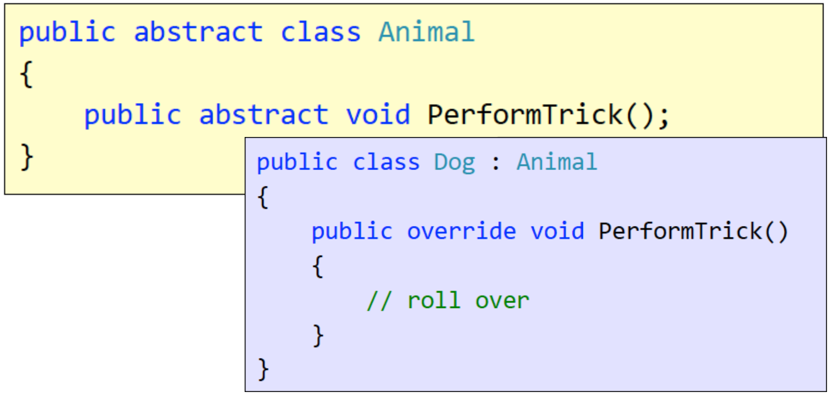
```
public class Dog : Animal
 {
     public override void PerformTrick()
     {
         // perform dog trick
         //  and then animal trick
         base.PerformTrick();
```

Class modifiers

- Abstract
- Static
- Sealed
- Partial

Abstract

- **The abstract keyword**
  - Can apply to a class
  - Can also apply to members (methods, properties, indexers, events)
- **Abstract class cannot be instantiated**
  - Abstract class is designed as a base class
  - Must implement abstract members to make a concrete class

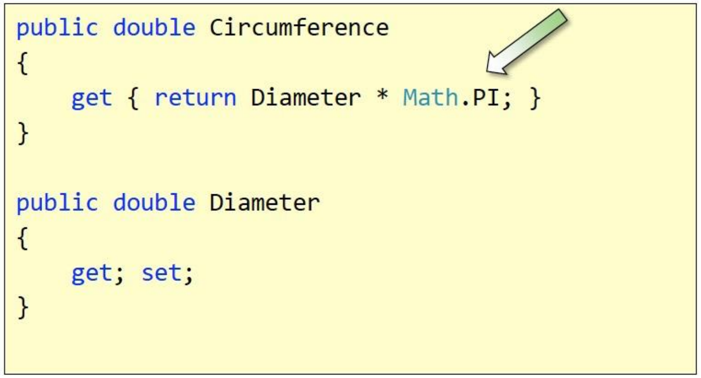


```
public abstract class Animal
{
    public abstract void PerformTrick();
}
```

```
public class Dog : Animal
{
    public override void PerformTrick()
    {
        // roll over
    }
}
```

Static

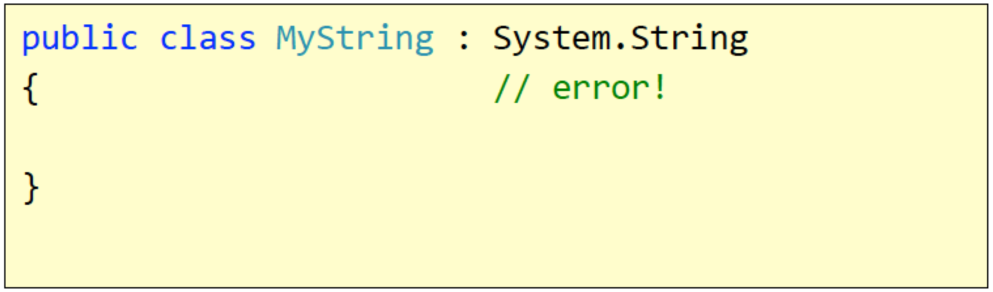
- **Static members are members of the type**
  - Cannot invoke the member through an object instance
- **Static classes can have only static members**
  - Cannot instantiate a static class
  - Used for “extension methods”



```
public double Circumference
{
    get { return Diameter * Math.PI; }
}

public double Diameter
{
    get; set;
}
```

Sealed

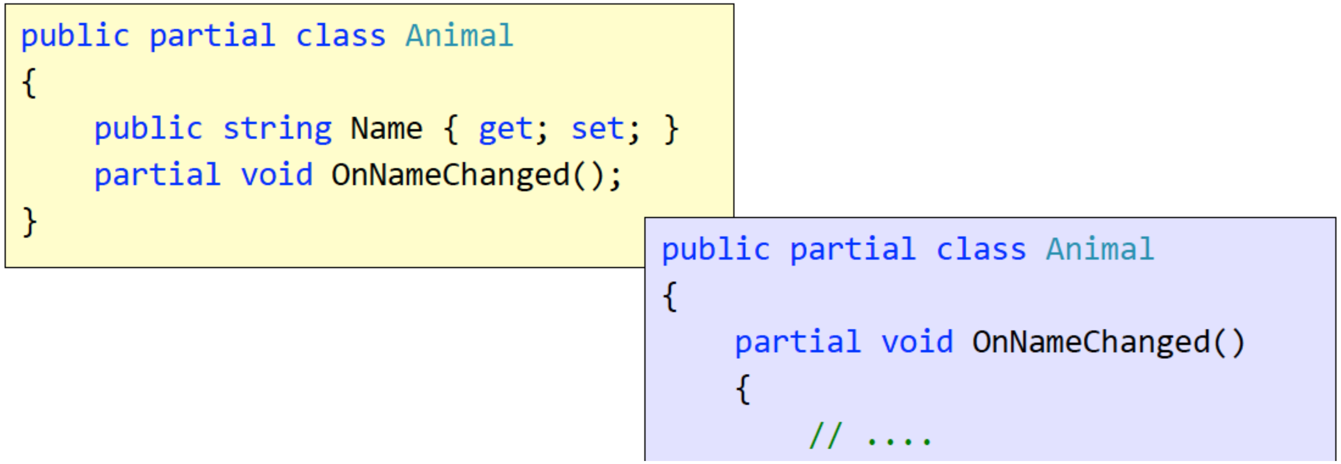
- **Sealed classes cannot be inherited**
  - They prevent extensibility or misuse
  - Some framework classes sealed for performance and security implications



```
public class MyString : System.String
{                           // error!

}
```

Partial classes

- **Partial classes are frequently generated by VS designer**
- **Partial class definitions can span multiple files**
  - But only in the same project
- **Partial method definitions are extensibility points**
  - Optimized away if no implementation provided



```
public partial class Animal
{
    public string Name { get; set; }
    partial void OnNameChanged();
}
```

```
public partial class Animal
{
    partial void OnNameChanged()
    {
        // ....
```

Summary

- **C# gives you everything you need for OOP**
  - Encapsulation
  - Inheritance
  - Polymorphism
- **Additional features for performance, convenience, extensibility**
  - Static classes
  - Sealed classes
  - Partial classes

Exercise / Demo

- Create the animal class hierarchy
- Create an abstract base class
- Create an abstract method: MakeSound
- Override MakeSound in each derived class and write to the console a unique sound for each
- Create an auto property Age of type int
- Initialize the Age property using a constructor
- Print the age property of an object to the console

.NET for C Programmers

Interfaces

Overview: Interfaces

- What is an interface?
- Defining an interface in C#
- Implementing an Interface
- Why use interfaces?

What are Interfaces?

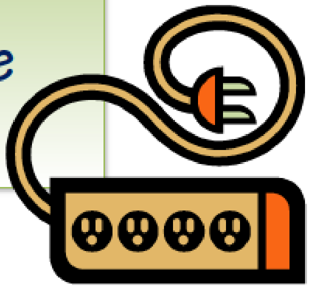
- Interfaces describe a group of related functions that can belong to any class or struct
- They provide a means of separating the client of an object from the implementation (we'll see more of this tomorrow)
- Public set of members:
  - Properties
  - Methods
  - Events
  - Indexers

Why Interfaces?

- Why would we use interfaces?

Maintainable

Extensible



Easily Testable

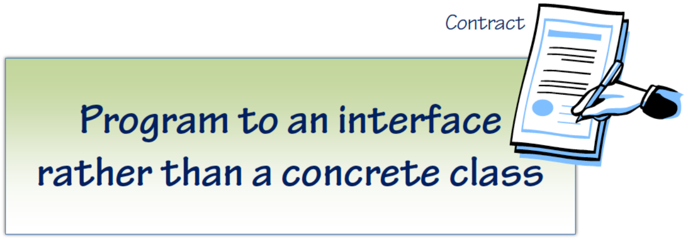
Best-Practice

- Coding to a concrete type causes issues with reuse and extensibility

Program to an abstraction rather than a concrete type

Best-Practice

- Code to interfaces instead of a concreate class, or even a class implemented abstractly
- User of the interface does not have knowledge of the type / implementation of the object behind it



Program to an interface rather than a concrete class

The Gist

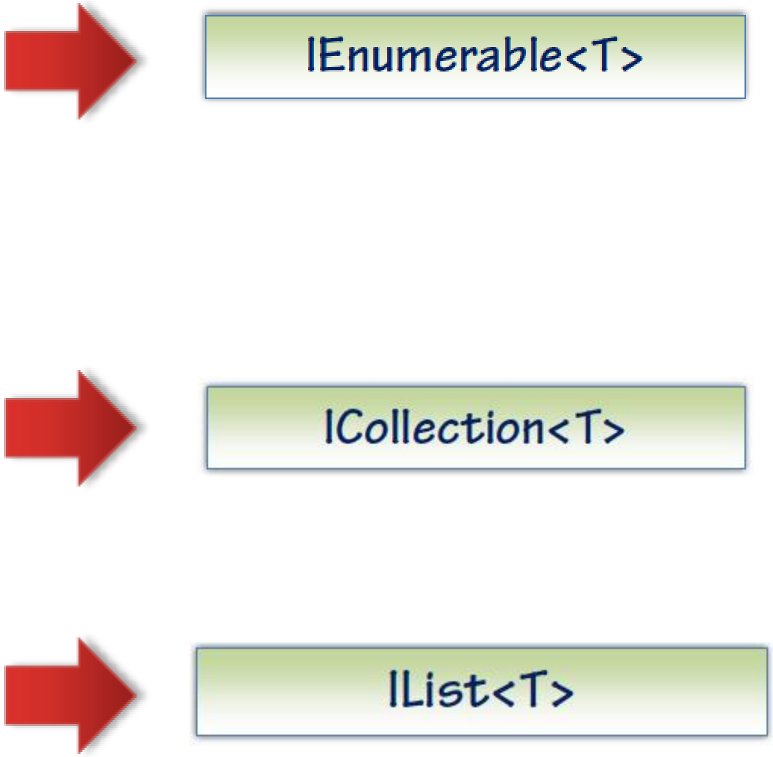
- **Concrete Class**
  - Brittle base class
- **Interface**
  - Resilience in the face of change
  - Insulation from implementation details

Example: Program at the right level

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

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

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



IEnumerable<T>

ICollection<T>

IList<T>

Interfaces and Abstract Classes in .NET

- **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**

Exercise

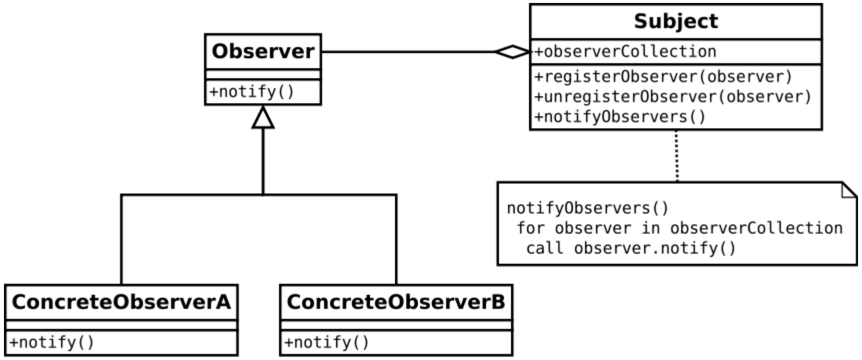
- Create interface IAnimal
- Define in interface property Age and method MakeSound
- Change derivation of Animal to use IAnimal

.NET for C Programmers

Events, Delegates and Event Handlers

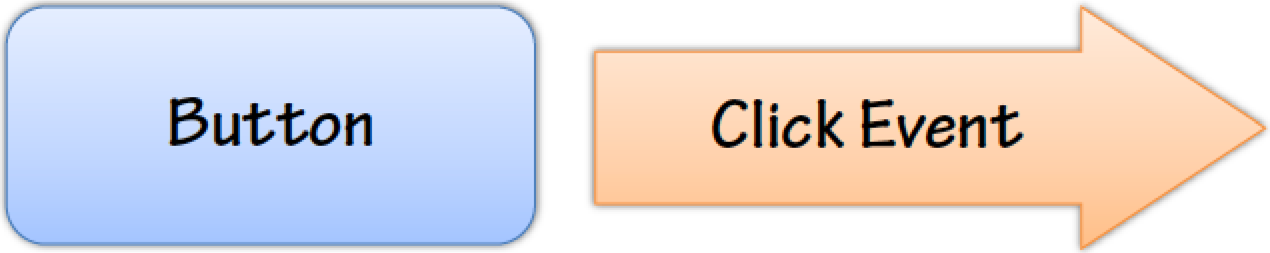
What is an event?

- **Events are notifications**
- **Play a central role in the .NET framework**
- **Provide a way to trigger notifications from end users or from objects**
- **.NET’s method of implementing the Observer pattern**
    - http://en.wikipedia.org/wiki/Observer_pattern



The Role of Events

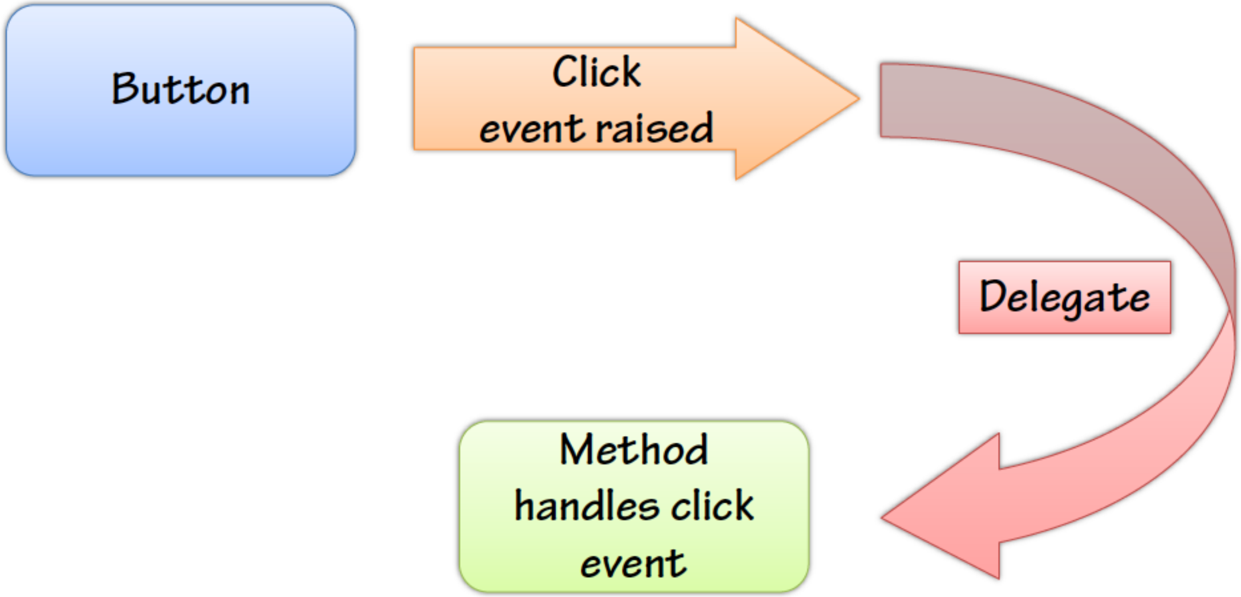
- Events signal the occurrence of an action/notification



- Objects that raise events don’t need to explicitly know the object that will handle the event
- Events pass EventArgs

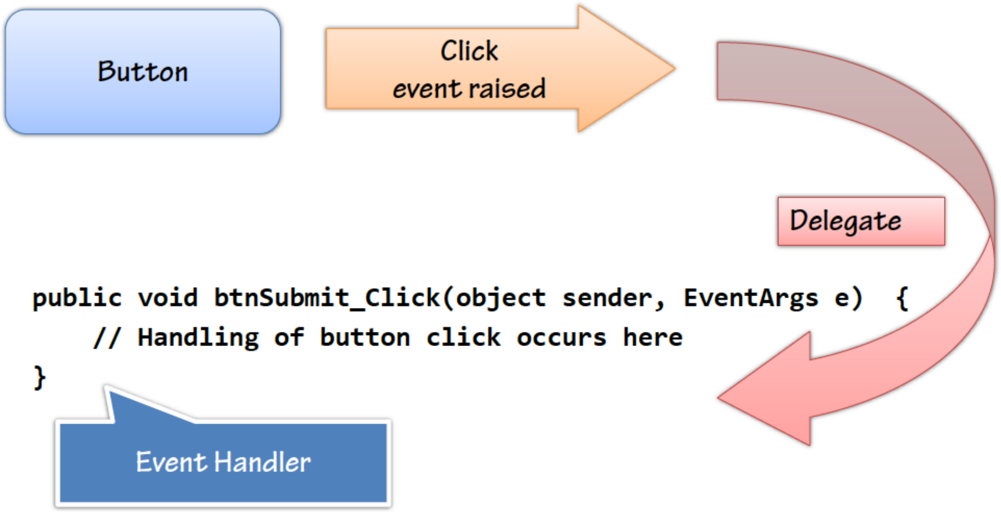
What is a Delegate?

- A specialized class that acts as a function pointer
- The means of routing an event to an event handler
- Based on a MulticaseDelegate base class



What is an Event Handler?

- Event handler is responsible for receiving and processing data from a delegate
- Normally receives two parameters:
  - Sender
  - EventArgs
- EventArgs encapsulate event data



```
public void btnSubmit_Click(object sender, EventArgs e)  {
    // Handling of button click occurs here
}
```

Exercise / Mid-class break

- Implement AgeChangedEvent in the animal class
  - Create a delegate for the event
  - Declare and Event based on the delegate
  - Add the event to the Animal class
  - Modify the Age property to call the delegate when the Age is changed via the set accessor
  - Create an object of a subclass of the Animal class
  - Connect a listener
  - Change the age

.NET for C Programmers

Change Notifications

Overview: Change Notifications

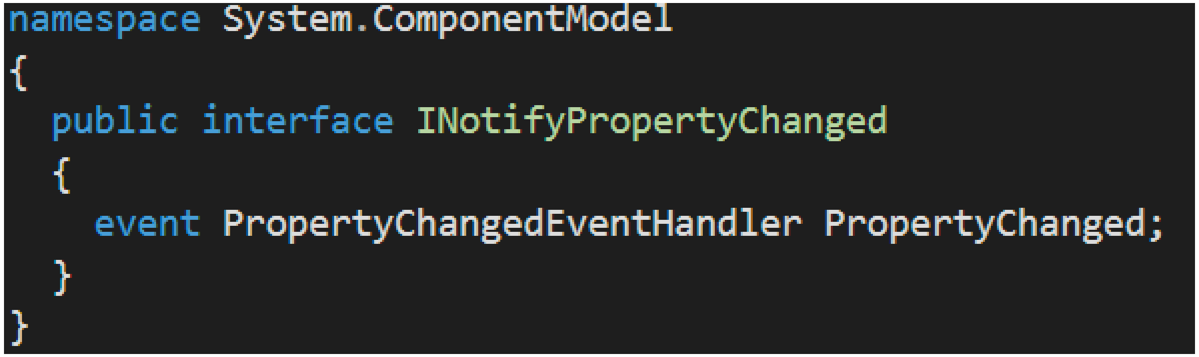
- Why use property notifications?
- INPC: INotifyPropertyChanged

Why?

- We often need to update user interfaces when data changes in an object, or
- A repository needs to track changes in objects so that it knows what to save / update.

How?

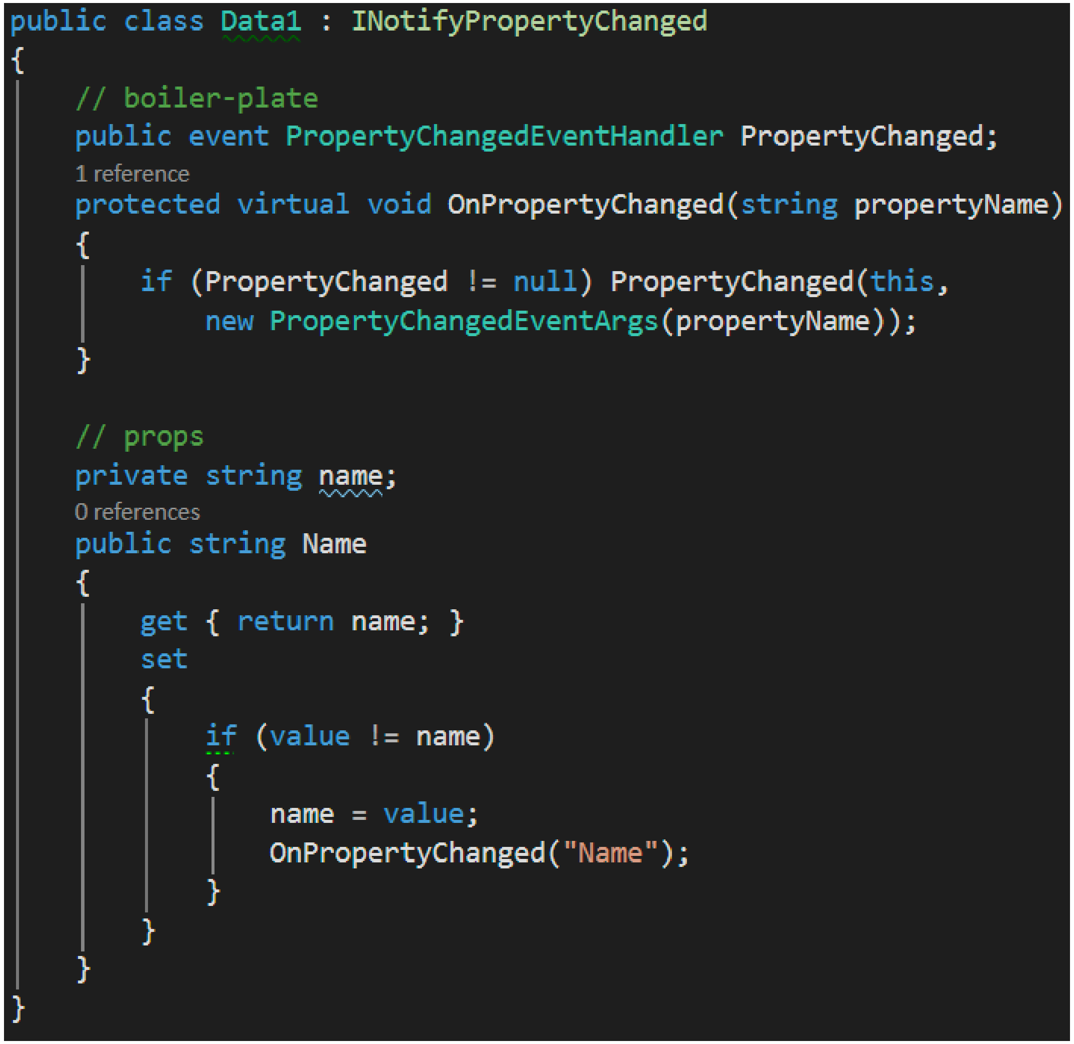
- Use the observer pattern
- In .Net, this is implemented using INPC
  - An interface, and
  - An implementation pattern in properties



```
namespace System.ComponentModel
{
  public interface INotifyPropertyChanged
  {
    event PropertyChangedEventHandler PropertyChanged;
  }
}
```

Common pattern of INPC

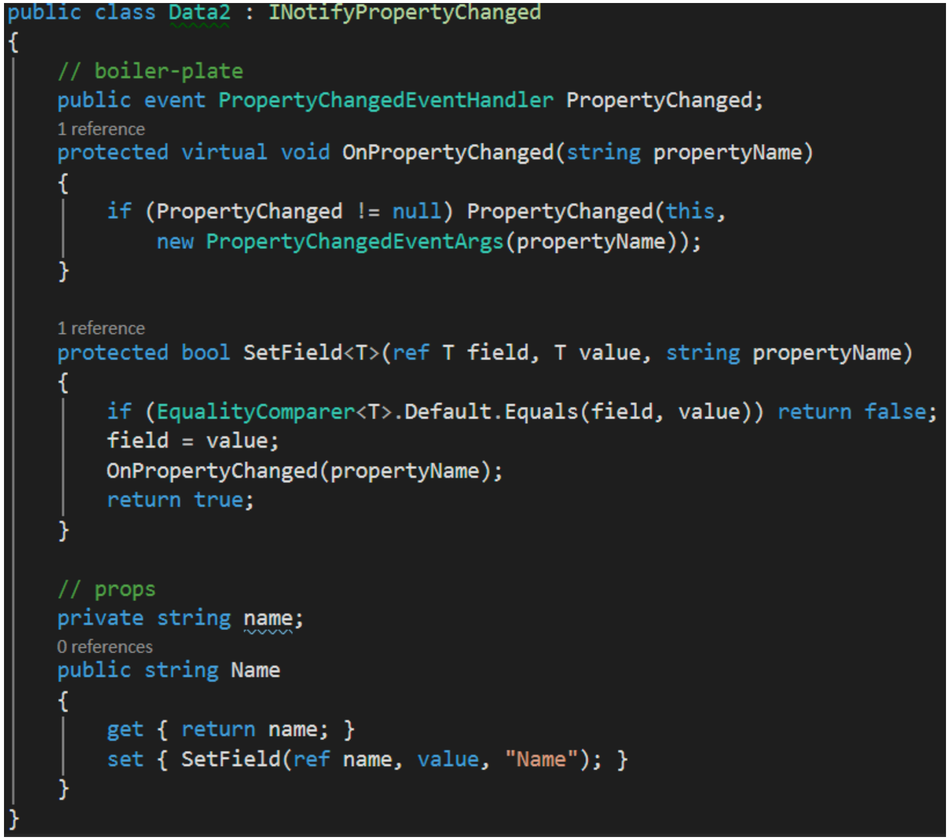
- Class implements INPC
- Which is only an event named PropertyChanged
- Then any properties needing can update using the pattern shown here



```
public class Data1 : INotifyPropertyChanged
{
    // boiler-plate
    public event PropertyChangedEventHandler PropertyChanged;
    1 reference
    protected virtual void OnPropertyChanged(string propertyName)
    {
        if (PropertyChanged != null) PropertyChanged(this,
            new PropertyChangedEventArgs(propertyName));
    }

    // props
    private string name;
    0 references
    public string Name
    {
        get { return name; }
        set
        {
            if (value != name)
            {
                name = value;
                OnPropertyChanged("Name");
            }
        }
    }
}
```

A Better Implementation of INPC

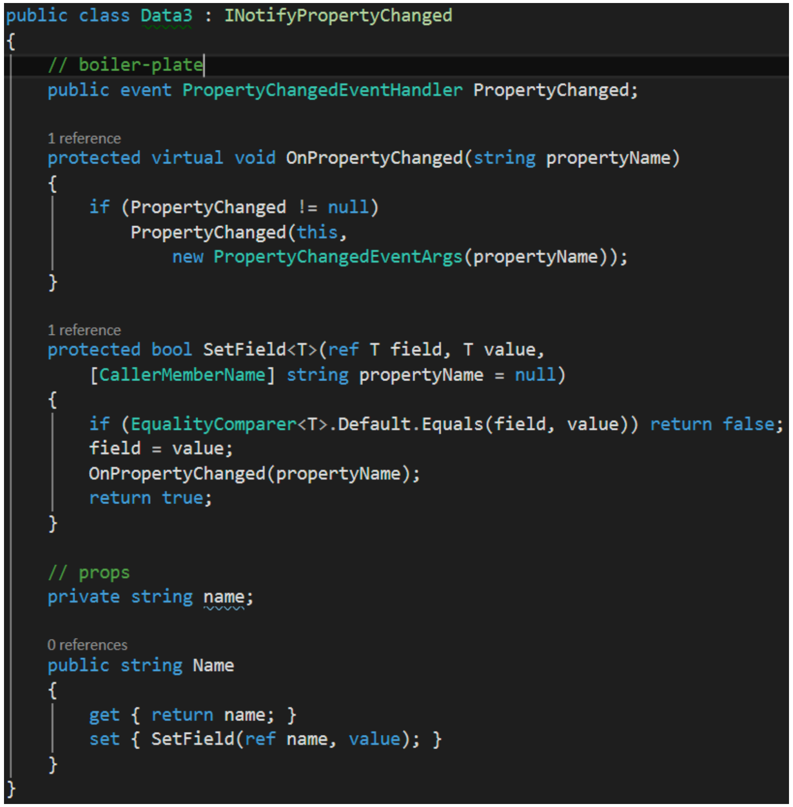
- Create a SetField protected method



```
public class Data2 : INotifyPropertyChanged
{
    // boiler-plate
    public event PropertyChangedEventHandler PropertyChanged;
    1 reference
    protected virtual void OnPropertyChanged(string propertyName)
    {
        if (PropertyChanged != null) PropertyChanged(this,
            new PropertyChangedEventArgs(propertyName));
    }

    1 reference
    protected bool SetField<T>(ref T field, T value, string propertyName)
    {
        if (EqualityComparer<T>.Default.Equals(field, value)) return false;
        field = value;
        OnPropertyChanged(propertyName);
        return true;
    }

    // props
    private string name;
    0 references
    public string Name
    {
        get { return name; }
        set { SetField(ref name, value, "Name"); }
    }
}
```

Even better with C# 5.0

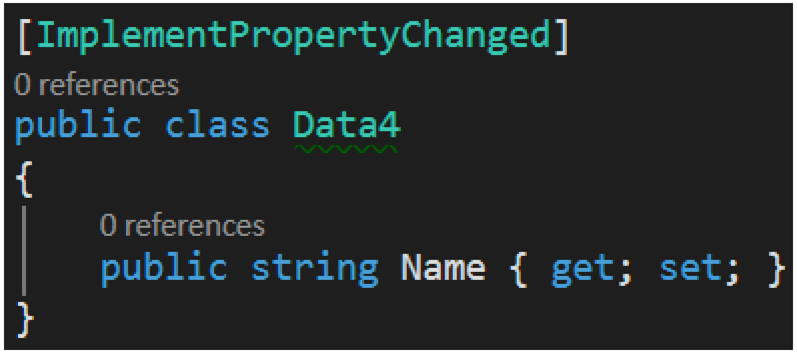
- Previous techniques are prone to coding in the wrong property name
- C# 5.0 solves this using the [CallerMemberName] attribute



```
public class Data3 : INotifyPropertyChanged
{
    // boiler-plate
    public event PropertyChangedEventHandler PropertyChanged;

    1 reference
    protected virtual void OnPropertyChanged(string propertyName)
    {
        if (PropertyChanged != null)
            PropertyChanged(this,
                new PropertyChangedEventArgs(propertyName));
    }

    1 reference
    protected bool SetField<T>(ref T field, T value,
        [CallerMemberName] string propertyName = null)
    {
        if (EqualityComparer<T>.Default.Equals(field, value)) return false;
        field = value;
        OnPropertyChanged(propertyName);
        return true;
    }

    // props
    private string name;

    0 references
    public string Name
    {
        get { return name; }
        set { SetField(ref name, value); }
    }
}
```

Best implementation of INPC with AoP

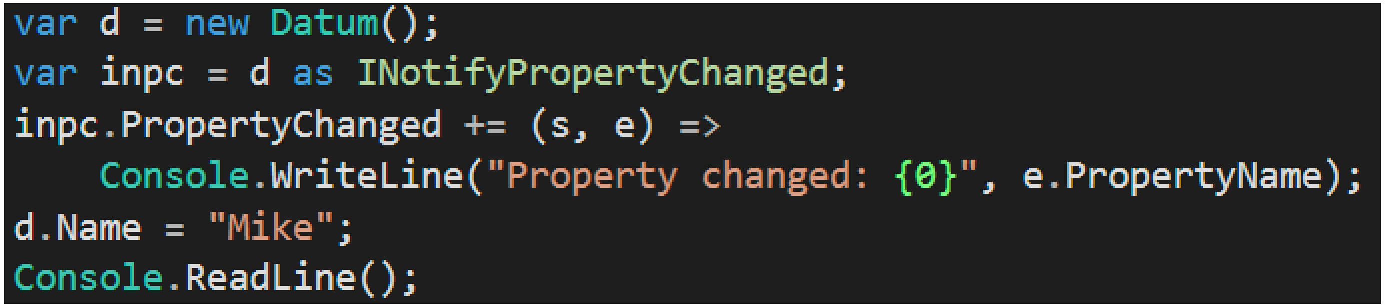
- This is using Fody AoP INPC weaver
- Any class with this attribute will have code injected to do all the INPC stuff
    - Forces the class to implement INPC
    - And rewrites the properties to do the right thing



```
[ImplementPropertyChanged]
0 references
public class Data4
{
    0 references
    public string Name { get; set; }
}
```

Observing

- Cast an object to INotifyPropertyChanged
- If not null
    - The object supports INPC
    - Subscribe to the event using an event handler



```
var d = new Datum();
var inpc = d as INotifyPropertyChanged;
inpc.PropertyChanged += (s, e) =>
    Console.WriteLine("Property changed: {0}", e.PropertyName);
d.Name = "Mike";
Console.ReadLine();
```

Actions, Funcs and lambdas

- In later version of .NET, delegates tend to get replaced by lamda functions
- Events are still useful as they are multicast and source of updates, where delegates/lamdas are the receivers
- lamdas are syntactically cleaner then delegates
- Action and Func types were added to C# allow lamdas to be passed to methods as parameters, or to define a field that is a lambda

Demo:

- Demonstration of using actions, lambdas and funcs

Exercise / Break

- Implement INPC in the age property of animal
- Implement an observer of animal
- Create an object of a subclass of animal
- Observe the animal
- Change the age

.NET for C Programmers

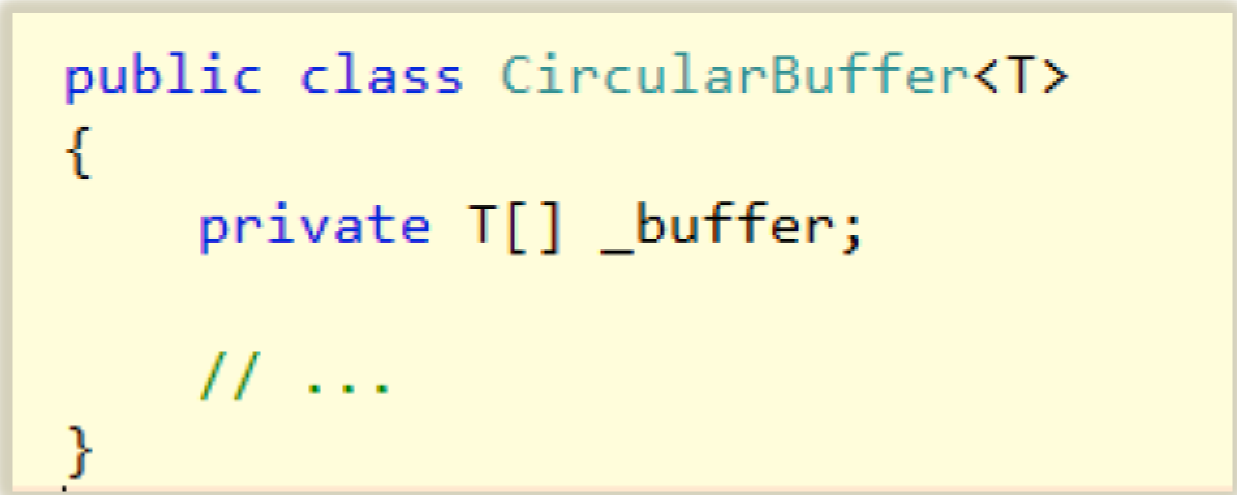
Generics

Overview

- Why
- Generic interfaces in .NET
- Generic methods
- Generic classes
- Constraints

Why

- **Generics types allow code reuse with type safety**
  - Defer type specification to client
  - Internal algorithms remain the same



```
public class CircularBuffer<T>
{
    private T[] _buffer;

    // ...
}
```

Generic Interfaces

Name	Purpose	Implemented By
IList<T>	Access by index	List<T>, SortedList<T>
ICollection<T>	Add, remove, and search	List<T> Dictionary<K, V> HashSet<T>
IDictionary<K, V>	Access by key	Dictionary<K,V>
IReadOnlyCollection<T>	Countable collection	List<T> Dictionary<K,V>
ISet<T>	Set based operations	HashSet<T>
IComparer<T>, IEqualityComparer<T>	Compare objects	

Generic Methods

- Methods can also be generic

Generic Constraints

- **Force a type parameter to have certain characteristics** Be a reference type or value type
  - Implement an interface
  - Derive from a base class
  - Have a default constructor
  - Be an instanced derive from another generic type parameter

Exercise

- Make the age property of Animal generic
  - Add <T> to the class
  - Change type of Age to T
  - Create a Dog<int> and Cat<T>
  - Print both Age properties to the console

.NET for C Programmers

Iteration

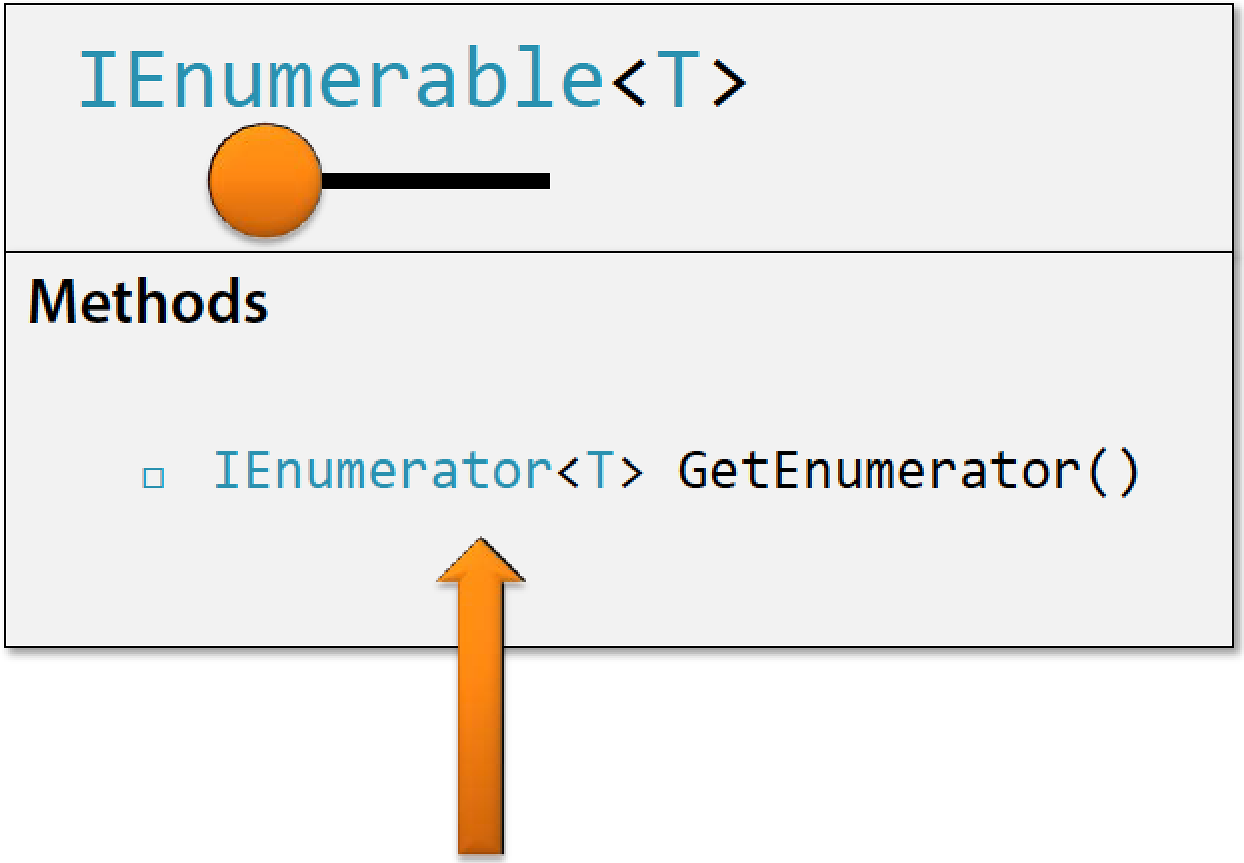
Overview

- What is iteration?
- IEnumerable<T> and IEnumerator<T>
- The foreach loop
- Enumerating collections that change
- Writing your own enumerators

What is iteration?

- It is the process of selecting, in-order, all objects in a collection
- The next module will focus on types of collections
- This one we focus on the process or iterating
  - Commonly referred to in .NET as Enumeration
- Enumeration requires two things:
  - An object provides an Enumerator via implementation of IEnumerable
  - Another method uses the enumerator to traverse the objects
- This is fundamentally core to .NET

Enumerating a collection



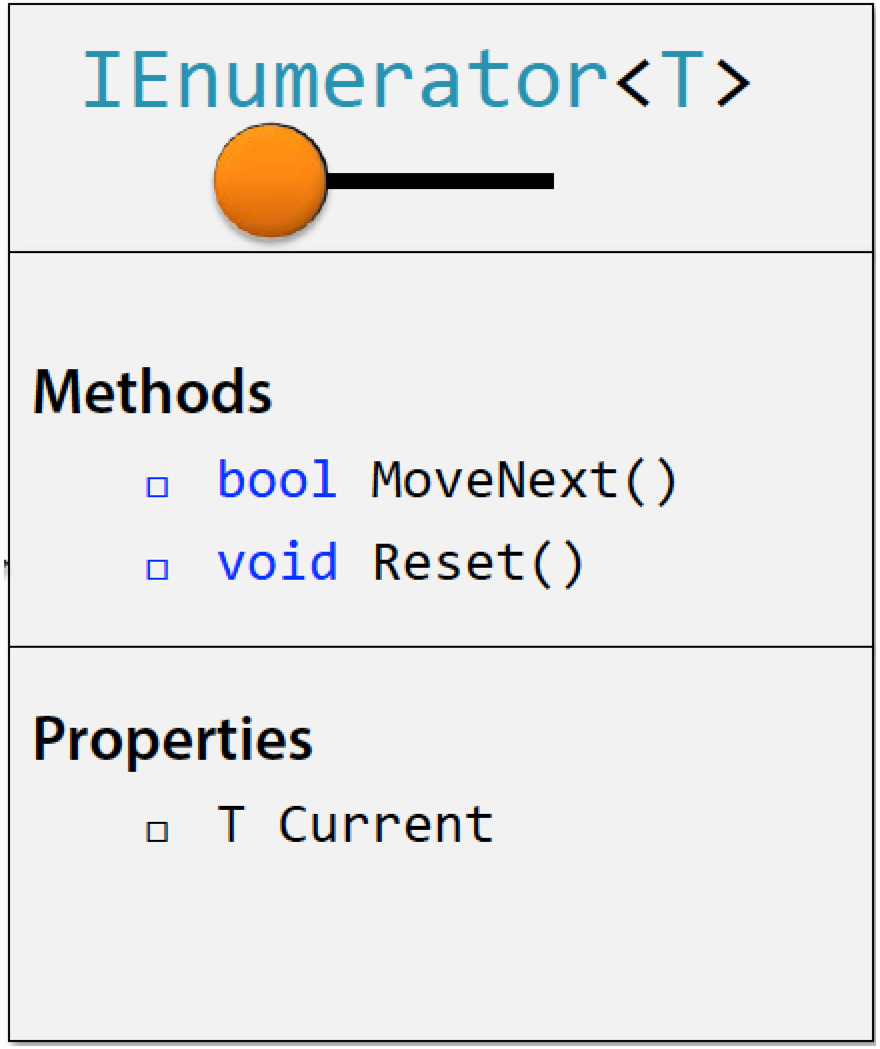
IEnumerable<T>

Methods

□ IEnumerator<T> GetEnumerator()

Use this method to get an enumerator to enumerate a collection

IEnumerator<T>



IEnumerator<T>

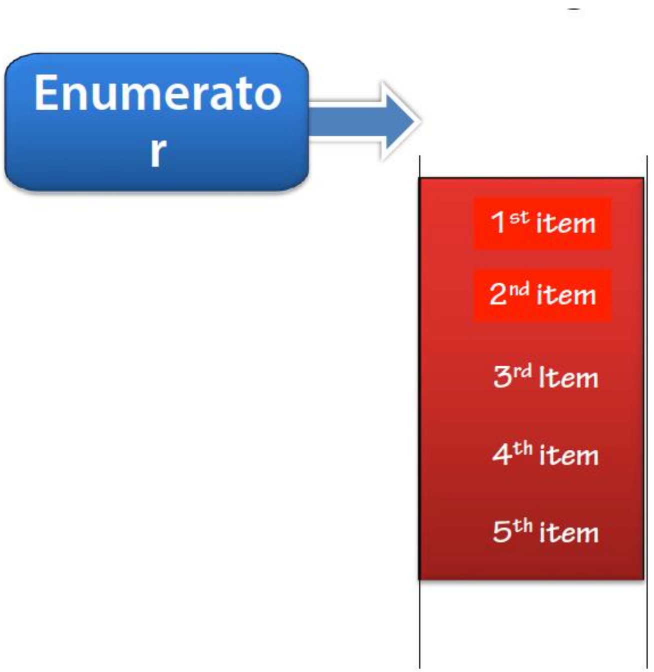
Methods

- □ `bool MoveNext()`
- □ `void Reset()`

Properties

- □ `T Current`

The process of enumeration



Enumerator

1st item

2nd item

3rd Item

4th item

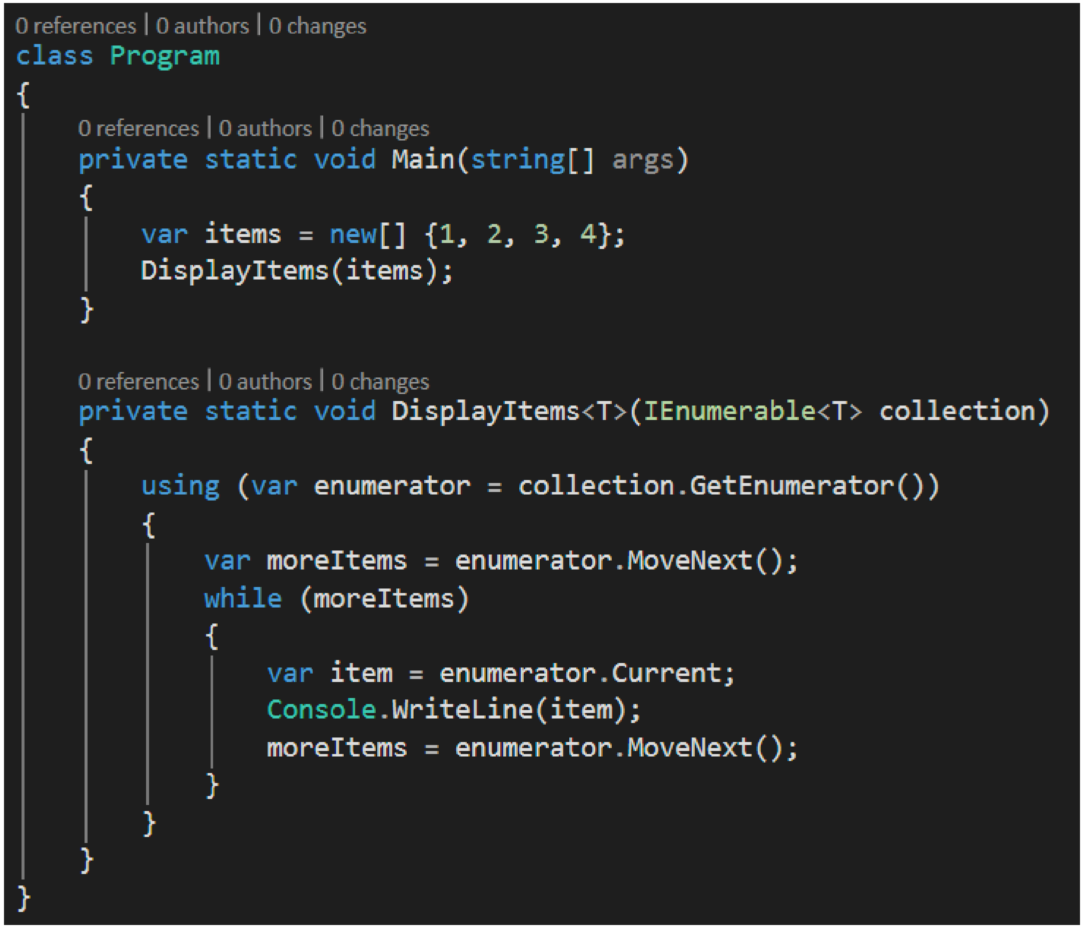
5th item

```
bool MoveNext();
```

True if there is at least one more item in the collection

The canonical iteration loop

- Create an array
- Pass it to a method as an Ienumerable
- Get the enumerator
- While MoveNext is true, loop and retrieve Current



```
0 references | 0 authors | 0 changes
class Program
{
    0 references | 0 authors | 0 changes
    private static void Main(string[] args)
    {
        var items = new[] {1, 2, 3, 4};
        DisplayItems(items);
    }

    0 references | 0 authors | 0 changes
    private static void DisplayItems<T>(IEnumerable<T> collection)
    {
        using (var enumerator = collection.GetEnumerator())
        {
            var moreItems = enumerator.MoveNext();
            while (moreItems)
            {
                var item = enumerator.Current;
                Console.WriteLine(item);
                moreItems = enumerator.MoveNext();
            }
        }
    }
}
```

The foreach loop

- The compiler implements the iteration loop
- But, if the collection is an array, the compiler will generate a for loop
    - More efficient

Why do we have Enumerables and Enumerators?

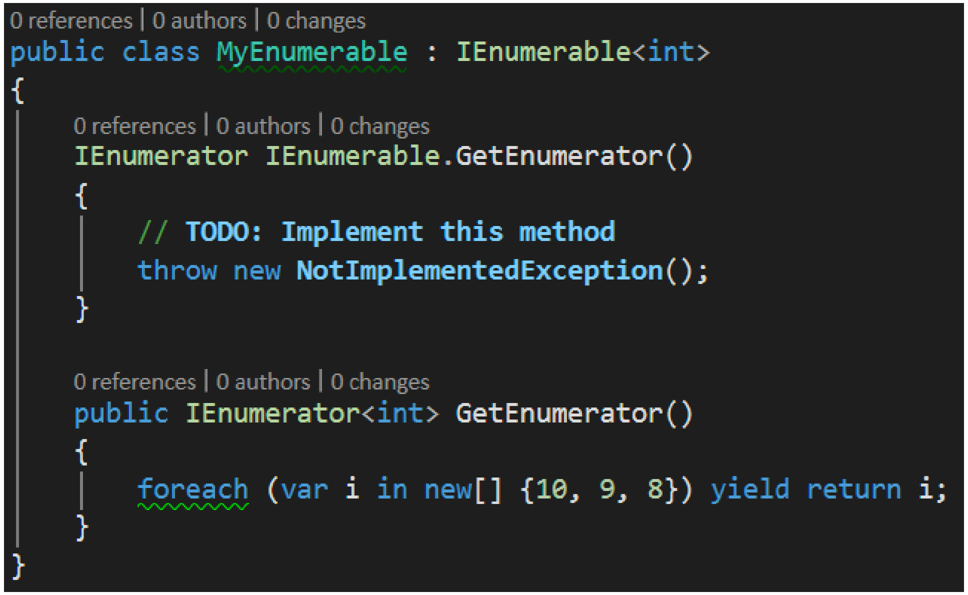
- A collection may need to have multiple enumerators running at the same time
- Each call to IEnumerable returns a different instance of Ienumerator
  - Allows more than one active enumeration across the data

Enumerating a collection that changes

- MoveNext will throw an exception if the collection has changed

Implementing IEnumerable

- This used to not be trivial
- Compiler support added with yield return to make it each
- Compiler auto-implements an implementation of IEnumerator for the type in the collection



```
0 references | 0 authors | 0 changes
public class MyEnumerable : IEnumerable<int>
{
    0 references | 0 authors | 0 changes
    IEnumerator IEnumerable.GetEnumerator()
    {
        // TODO: Implement this method
        throw new NotImplementedException();
    }

    0 references | 0 authors | 0 changes
    public IEnumerator<int> GetEnumerator()
    {
        foreach (var i in new[] {10, 9, 8}) yield return i;
    }
}
```

.NET for C Programmers

Collections

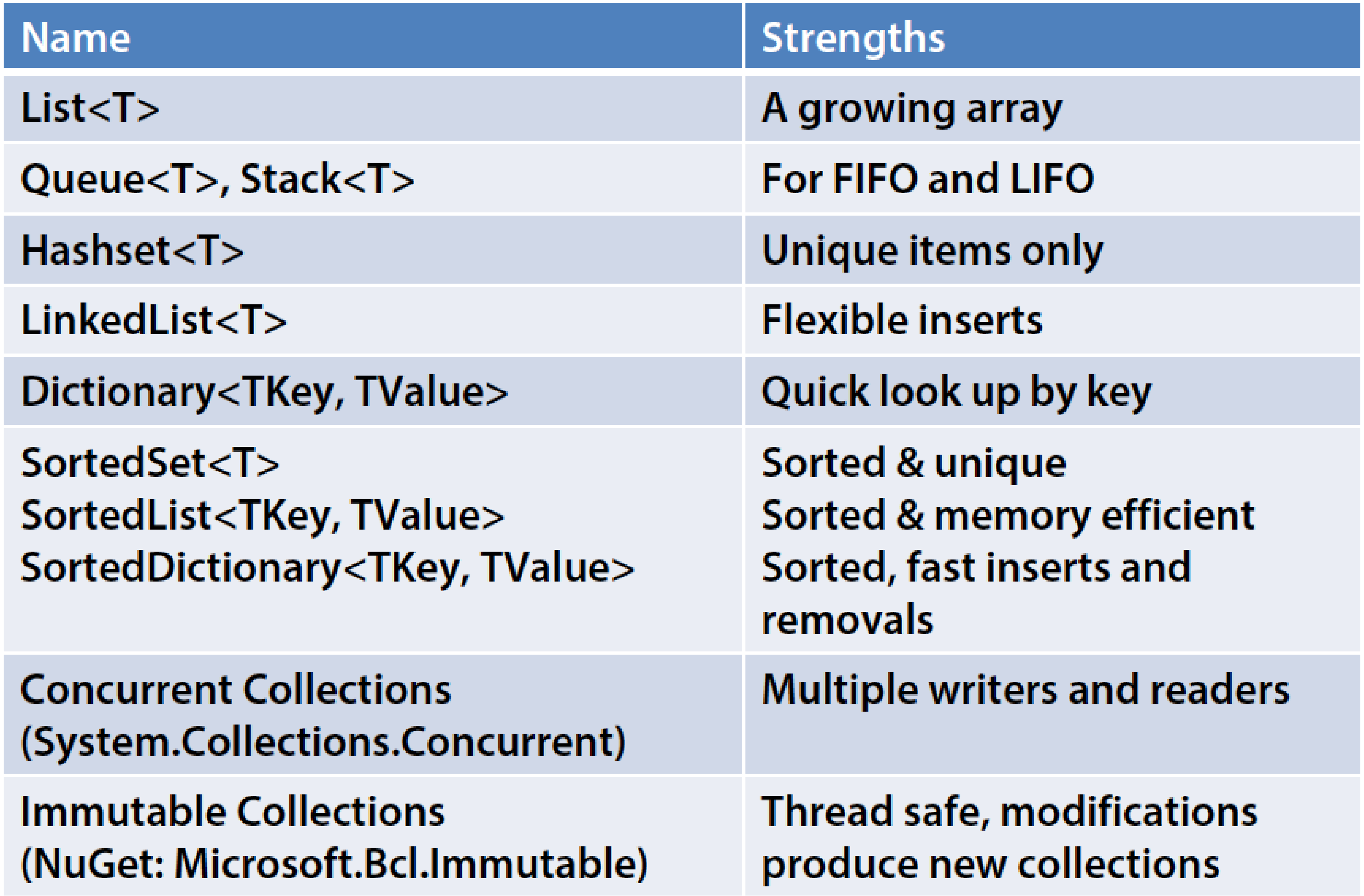
Overview

- What is a collection?
- Collections in .NET
- Arrays
- Collection Interfaces
- Collection Types
    - Index based lists
    - Other lists
    - Dictionaries
    - Sets
- Enumerators

What is a collection?

- They are a group of related objects
  - Any array is a collection
- But they are more flexible than arrays
- Items can be looked up by index or key
- Storage may not be contiguous

.NET has many collection classes



Name	Strengths
List<T>	A growing array
Queue<T>, Stack<T>	For FIFO and LIFO
Hashset<T>	Unique items only
LinkedList<T>	Flexible inserts
Dictionary<TKey, TValue>	Quick look up by key
SortedSet<T> SortedList<TKey, TValue> SortedDictionary<TKey, TValue>	Sorted & unique Sorted & memory efficient Sorted, fast inserts and removals
Concurrent Collections (System.Collections.Concurrent)	Multiple writers and readers
Immutable Collections (NuGet: Microsoft.Bcl.Immutable)	Thread safe, modifications produce new collections

These classes derive from many types of base classes and interfaces

ReadOnlyCollection<T>

SortedList<TKey, TValue>

LinkedList<T>

Stack<T>

ObservableCollection<T>

HashSet<T>

Array

List<T>

SortedSet<T>

Collection<T>

Dictionary<TKey, TValue>

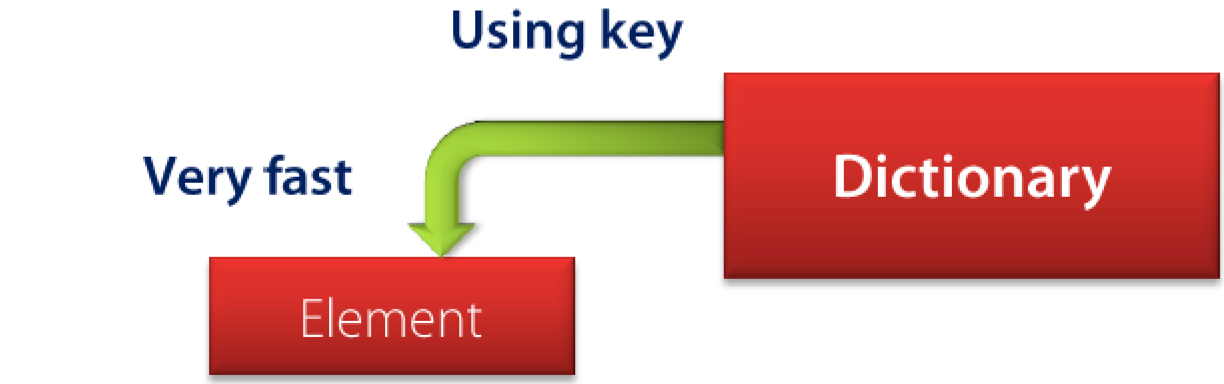
KeyedCollection<TKey, TItem>

Lists

Dictionaries

Sets

Looking up an element, Dict/List

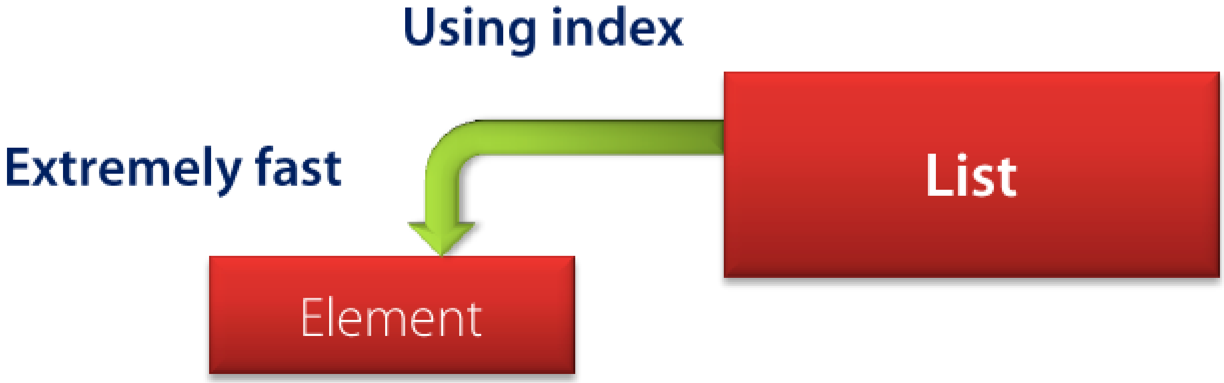


Using key

Very fast

Dictionary

Element



Using index

Extremely fast

List

Element

Enumeration vs Lookup

- All .Net collections implement IEnumerable
- But may implement other collection based interfaces
    - ICollection, IList, …
- Enumeration will travers each object
- Lookup gets a single element

Enumerating

All collections

Looking up items

Many collections

NOT: Sets

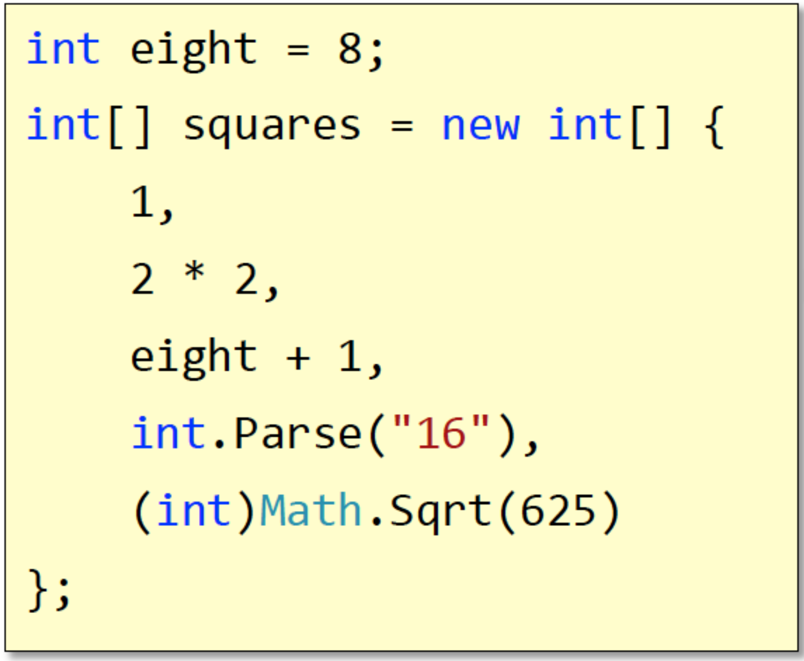
NOT: Linked lists, Stacks, Queues

Arrays

- Single memory block
- Indexed by integers stating at 0
- Fixed in size
- Multi-dimensional
- Can be Jagged
- Consist of either value or reference types

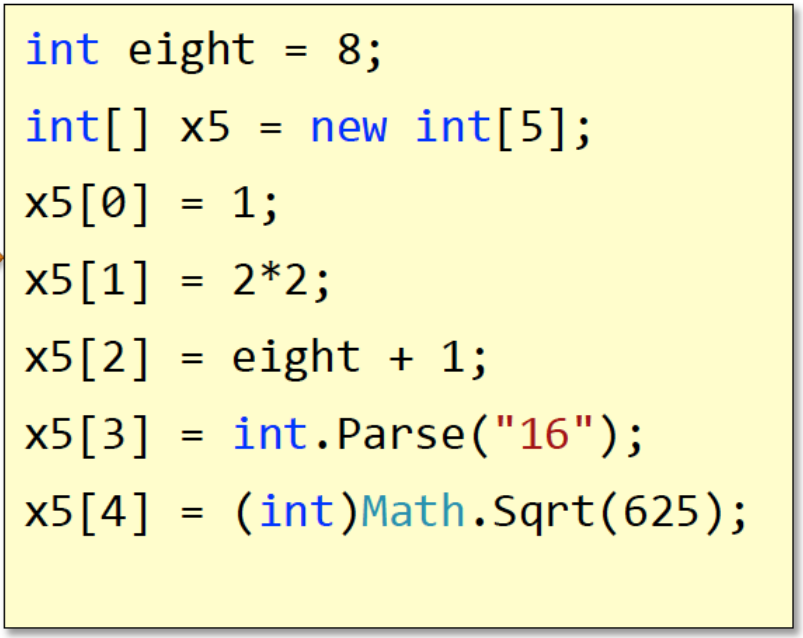
Array Initializers

Compiler turns this...



```
int eight = 8;
int[] squares = new int[] {
    1,
    2 * 2,
    eight + 1,
    int.Parse("16"),
    (int)Math.Sqrt(625)
};
```

...into (roughly) this



```
int eight = 8;
int[] x5 = new int[5];
x5[0] = 1;
x5[1] = 2*2;
x5[2] = eight + 1;
x5[3] = int.Parse("16");
x5[4] = (int)Math.Sqrt(625);
```

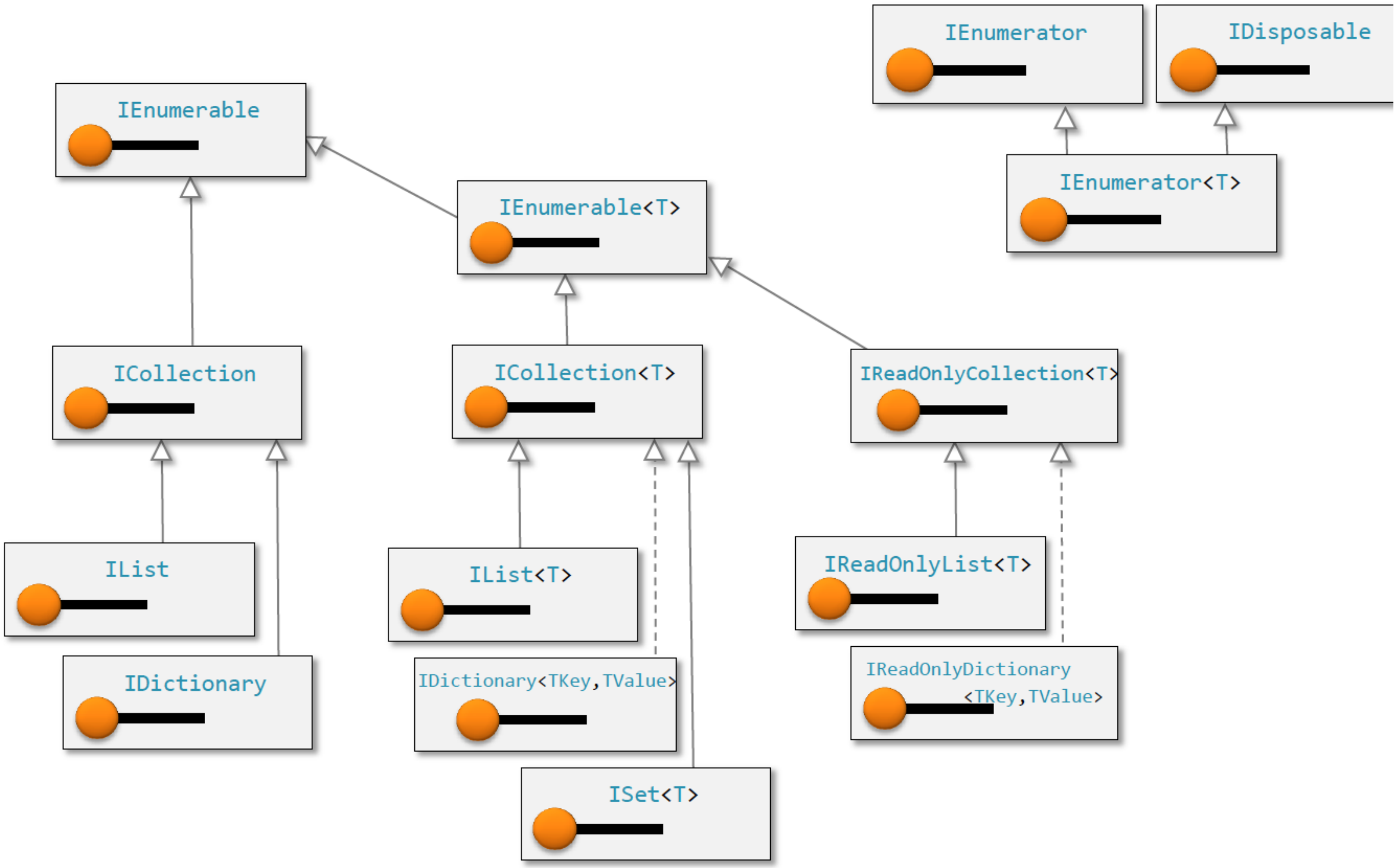
Arrays Implement IEnumerable

- Used for
  - foreach
  - LINQ

The Array class

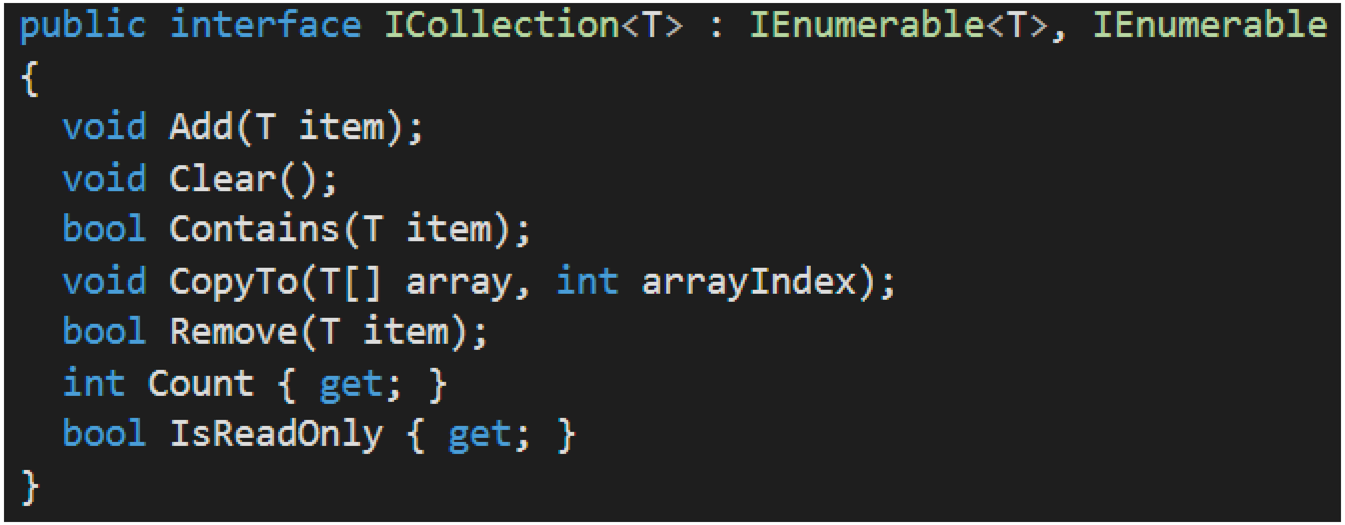
- Provides a set of methods for advanced manipulation of arrays, instead of simple position or enumeration
- Examples
  - Array.BinarySearch
  - Array.FindIndex
  - …

Collection Interfaces



ICollection

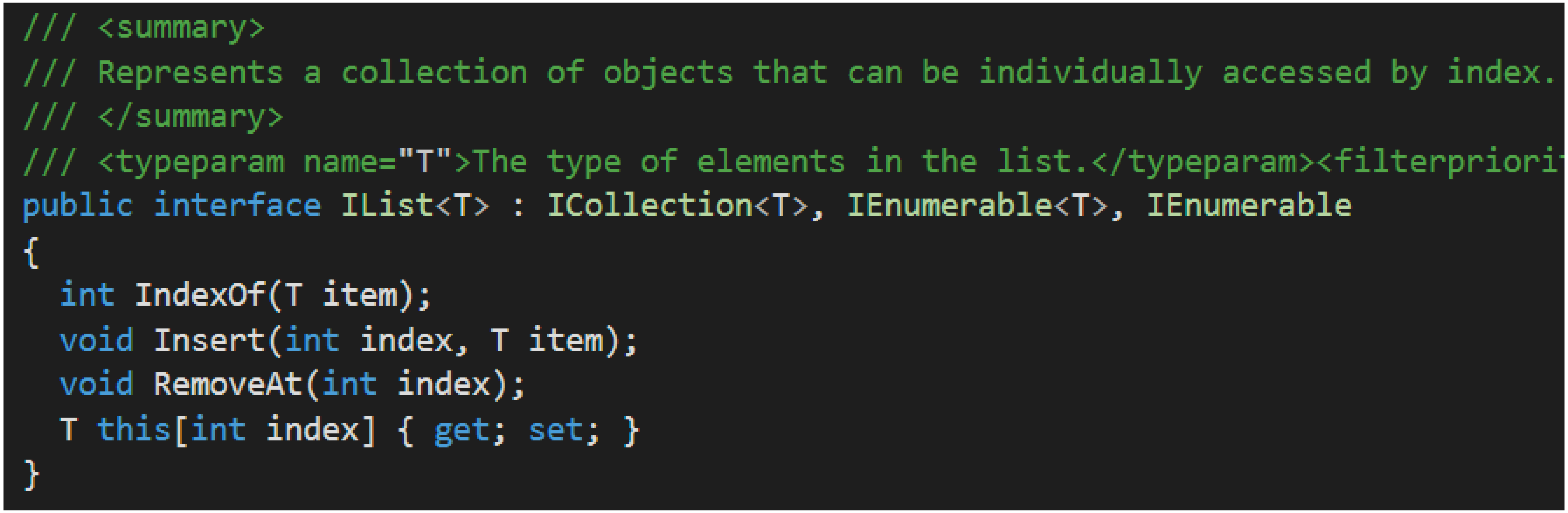
- A collection “says”:
  - I know how many elements I have
  - You can modify my contents



```
public interface ICollection<T> : IEnumerable<T>, IEnumerable
{
  void Add(T item);
  void Clear();
  bool Contains(T item);
  void CopyTo(T[] array, int arrayIndex);
  bool Remove(T item);
  int Count { get; }
  bool IsReadOnly { get; }
}
```

IList<T>

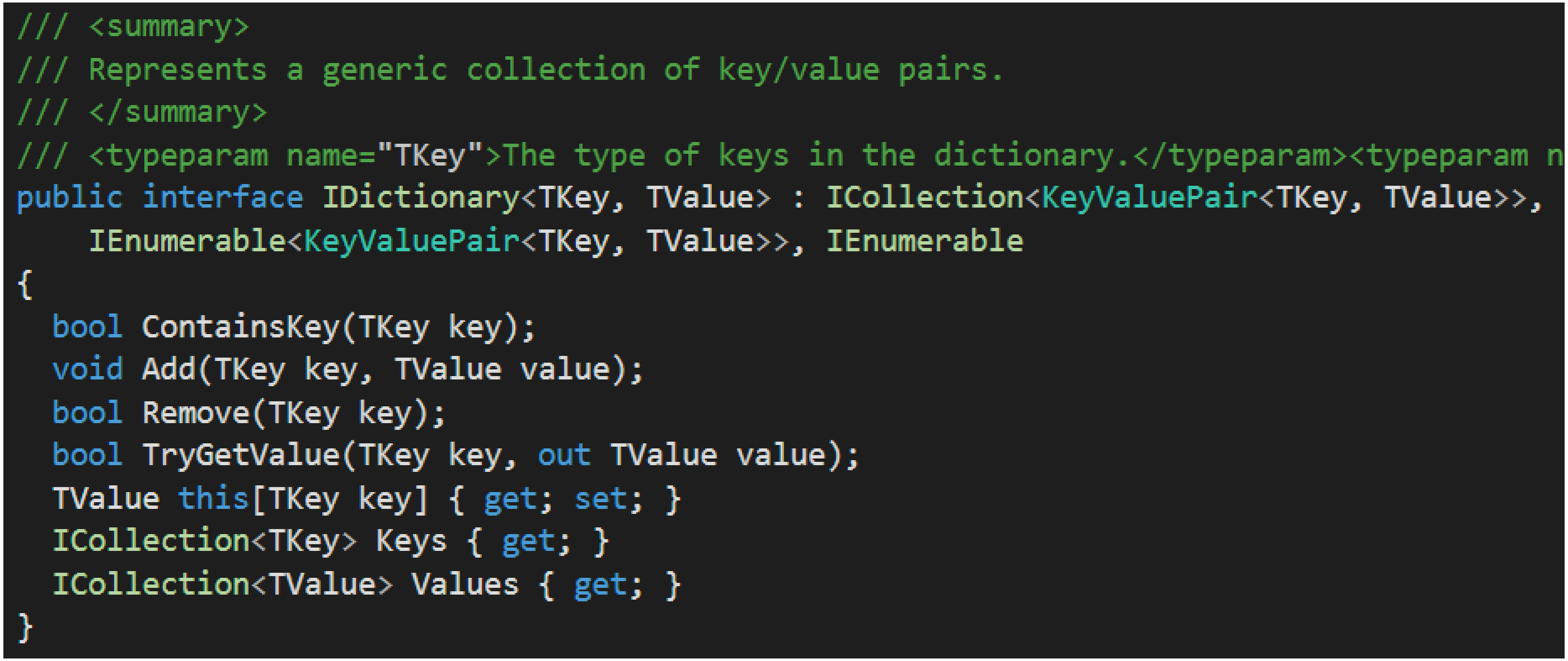
- Items can be accessed by index



```
/// <summary>
/// Represents a collection of objects that can be individually accessed by index.
/// </summary>
/// <typeparam name="T">The type of elements in the list.</typeparam><filterpriori
public interface IList<T> : ICollection<T>, IEnumerable<T>, IEnumerable
{
  int IndexOf(T item);
  void Insert(int index, T item);
  void RemoveAt(int index);
  T this[int index] { get; set; }
}
```

IDictionary<K, V>

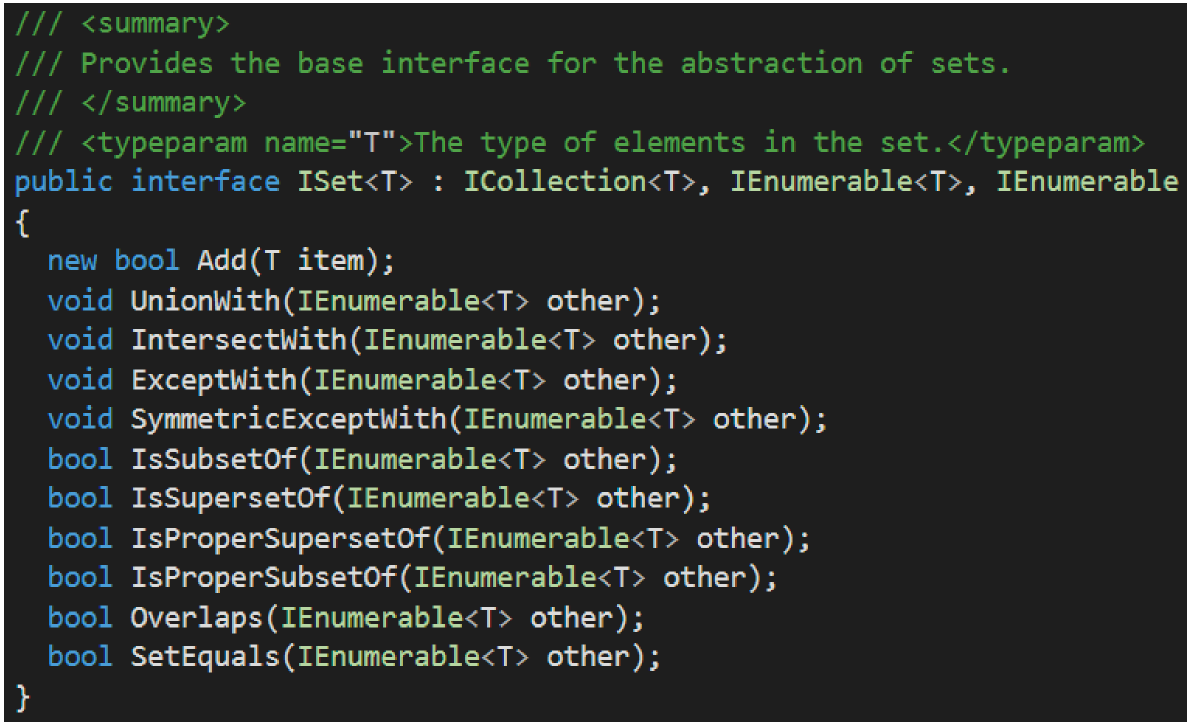
- You can look up my elements with a key



```
/// <summary>
/// Represents a generic collection of key/value pairs.
/// </summary>
/// <typeparam name="TKey">The type of keys in the dictionary.</typeparam><typeparam n
public interface IDictionary<TKey, TValue> : ICollection<KeyValuePair<TKey, TValue>>,
    IEnumerable<KeyValuePair<TKey, TValue>>, IEnumerable
{
  bool ContainsKey(TKey key);
  void Add(TKey key, TValue value);
  bool Remove(TKey key);
  bool TryGetValue(TKey key, out TValue value);
  TValue this[TKey key] { get; set; }
  ICollection<TKey> Keys { get; }
  ICollection<TValue> Values { get; }
}
```

ISet<T>

- I can do set operations with other collections
- I cannot have duplicate items



```
/// <summary>
/// Provides the base interface for the abstraction of sets.
/// </summary>
/// <typeparam name="T">The type of elements in the set.</typeparam>
public interface ISet<T> : ICollection<T>, IEnumerable<T>, IEnumerable
{
  new bool Add(T item);
  void UnionWith(IEnumerable<T> other);
  void IntersectWith(IEnumerable<T> other);
  void ExceptWith(IEnumerable<T> other);
  void SymmetricExceptWith(IEnumerable<T> other);
  bool IsSubsetOf(IEnumerable<T> other);
  bool IsSupersetOf(IEnumerable<T> other);
  bool IsProperSupersetOf(IEnumerable<T> other);
  bool IsProperSubsetOf(IEnumerable<T> other);
  bool Overlaps(IEnumerable<T> other);
  bool SetEquals(IEnumerable<T> other);
}
```

List<T>

- Probably the most common collection in .NET (other than array)
- Encapsulates an array T[]
- Generally high performance as elements can be stored in a contiguous block of memory
- Provides for integer based lookup like an array
- Resizable
    - items can be inserted in-between and at the end

LinkedList<T> / LinkedListNode<T>

- Non-index based list
- Fast adding / removing of elements
- Each item in the list is an instance of LinkedListNode

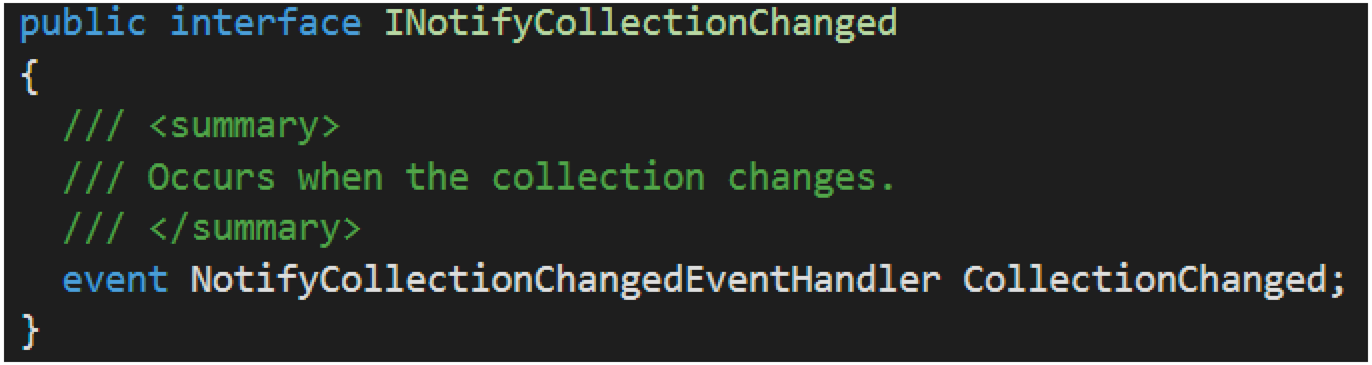
Collection<T>

- Allows the customizing the behavior of List<T>

- Such as…

ObservableCollection<T>

- Gives collection changed notifications
- INotifyCollectionChanged



```
public interface INotifyCollectionChanged
{
  /// <summary>
  /// Occurs when the collection changes.
  /// </summary>
  event NotifyCollectionChangedEventHandler CollectionChanged;
}
```

Stack<T> / Queue<T>

- LIFO / FIFO operation

Dictionaries

- Dictionary<K,V>
- SortedList<K,V>
- SortedDictionary<K,V>
- KeyedCollection<K,V>

Sets

- HashSet<T>
- SortedSet<T>

Exercises

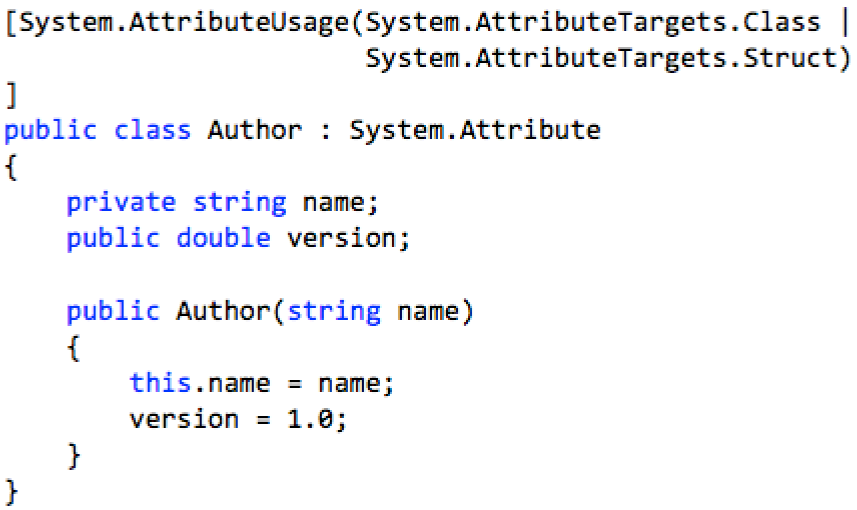
- Create and enumerate a list
- Create and lookup items in a dictionary

.NET for C Programmers

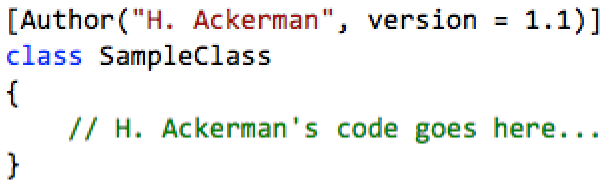
Attributes

Attributes are…

- Custom metadata
- Can be applied to
  - Classes
  - Methods
  - Parameters
- Accessed via reflection



```
[System.AttributeUsage(System.AttributeTargets.Class |
                       System.AttributeTargets.Struct)
]
public class Author : System.Attribute
{
    private string name;
    public double version;

    public Author(string name)
    {
        this.name = name;
        version = 1.0;
    }
}
```



```
[Author("H. Ackerman", version = 1.1)]
class SampleClass
{
    // H. Ackerman's code goes here...
}
```

Demo: Attributes in metadata