Advanced Programming Methods
Lecture 11 – More on C#

C# GENERICS

C#'s genericity mechanism, available since C# 2.0

Most common use:

- Use (and implement) generic yet type-safe containers

```
List<String> safeBox = new List<String>();
```

Compile-time type-checking is enforced

- Custom generic classes (and methods)

Syntax: different position of the generic parameter w.r.t. Java

Java: `public <T> T foo (T x);`

C#: `public T foo <T> (T x);`

- Bounded genericity

```
public T test <T> (T x) where T:Interface1, Interface2
```

C# vs Java

- Unlike Java, genericity is supported natively by .NET bytecode

- Hence, basically all limitations of Java generics disappear:
  - Can instantiate generic parameter with value types
  - At runtime you can tell the difference between `List<Integer>` and `List<String>`
  - Exception classes can be generic classes
  - Can instantiate a generic type parameter provided a clause `where G : new()` constrains the parameter to have a default constructor

C# vs. Java

- Can get the default value of a generic type parameter
  `T t = default (T);`
- Arrays can have elements of a generic type parameter
- A static member can reference a generic type parameter

Another consequence is that **raw types** (unchecked generic types without any type argument) don't exist in C#

Generics and Inheritance

- Let S be a subtype of T
- There is no inheritance relation between:
  `SomeGenericClass<S>` and **`SomeGenericClass<T>`**

  In particular: the former is not a subtype of the latter

- However, let AClass be a non-generic type:

  `S<AClass>` is a subtype of **`T<AClass>`**

Replacing Wildcards in C#

- There's no C# equivalent of Java's wildcards

  - But most of Java's wildcard code can be ported to C# (not necessarily resulting in cleaner code)

Consider the following hierarchy of classes:

class Circle:Shapes{...}

class Rectangle:Shapes{...}

What should be the signature of a method **`drawShapes`** that takes a list of **`Shape`** objects and draws all of them?

DrawShapes(List<Shapes> shapes)

- **this doesn't work on a `List<Circle>`, which is not a subtype of `List<Shape>`**

Replacing Wildcards in C#

Solution: use a helper class with bounded genericity

```
class DrawHelper <T> where T: Shape {
public static void DrawShapes( List<T> shapes);
}
The use of the method:

DrawHelper<Shape>.DrawShapes(listOfShapes);
DrawHelper<Circle>.DrawShapes(listOfCircles);
```

Generics can be used with:

- Types
  - Struct
  - Interface
  - Class
  - Delegate
- Methods
- Generic Constraints

Generics can be used:

- to easily create non-generic derived types:

```
public class IntStack : Stack<int> {...}
```

- in internal fields, properties and methods of a class:

```
public class Customer<T>{
  private static List<T> customerList;
  private T customerInfo;
  public T CustomerInfo { get; set; }
  public int CompareCustomers( T customerInfo );}
```

- A base class or interface can be used as a constraint.For instance:

```
public interface IDrawable { public void Draw(); }
```

  - Need a constraint that our type T implements the IDrawable interface.

```
public class SceneGraph<T> where T : IDrawable {
    public void Render() { … T node; …
        node.Draw();
    }
  }
```

- No need to cast
  - Compiler uses type information to decide

- Can also specify a class constraint. That is, require a reference type:

```
public class CarFactory<T> where T : class {
private T currentCar = null;
```

  - Forbids CarFactory<int> and other value types.
  - Useful since I can not set an int to null.

- Alternatively, require a value (struct) type.

```
public struct Nullable<T> where T : struct {
private T value;
```

- The ***default*** keyword

```
public class GraphNode<T> {
  private T nodeLabel;
  private void ClearLabel() { nodeLabel = default(T); }
```

  - If T is a reference type default(T) will be null.
  - For value types all bits are set to zero.

- Special constraint using the *new* keyword:

```
public class Stack<T> where T : new() {
    public T PopEmpty() {
        return new T();
    }
}
```

- Parameter-less ***constructor constraint***
- Type T must provide a public parameter-less constructor
- No support for other constructors or other method syntaxes.
- The new() constraint must be the last constraint.

- A generic type parameter, like a regular type, can have zero or more interface constraints

```
public class GraphNode<T> {
    where T : ICloneable, IComparable
        ...}
```

- A type parameter can only have one where clause, so all constraints must be specified within a single where clause.
- You can also have one type parameter be dependent on another.

```
public class SubSet<U,V> where U : V
public class Group<U,V>
    where V : IEnumerable<U>  { … }
```

- C# also allows you to parameterize a method with generic types:

```
public static void Swap<T>( ref T a, ref T b ){
    T temp = a;
    a = b;
    b = temp; }
```

- The method does not need to be static.

```
public class Report<T> : where T: Iformatter{...}
public class Insurance {
   public Report<T> ProduceReport<T>() where T : Iformatter  {…}
}
```

- In C#, generic types can be compiled into a class library or dll and used by many applications.

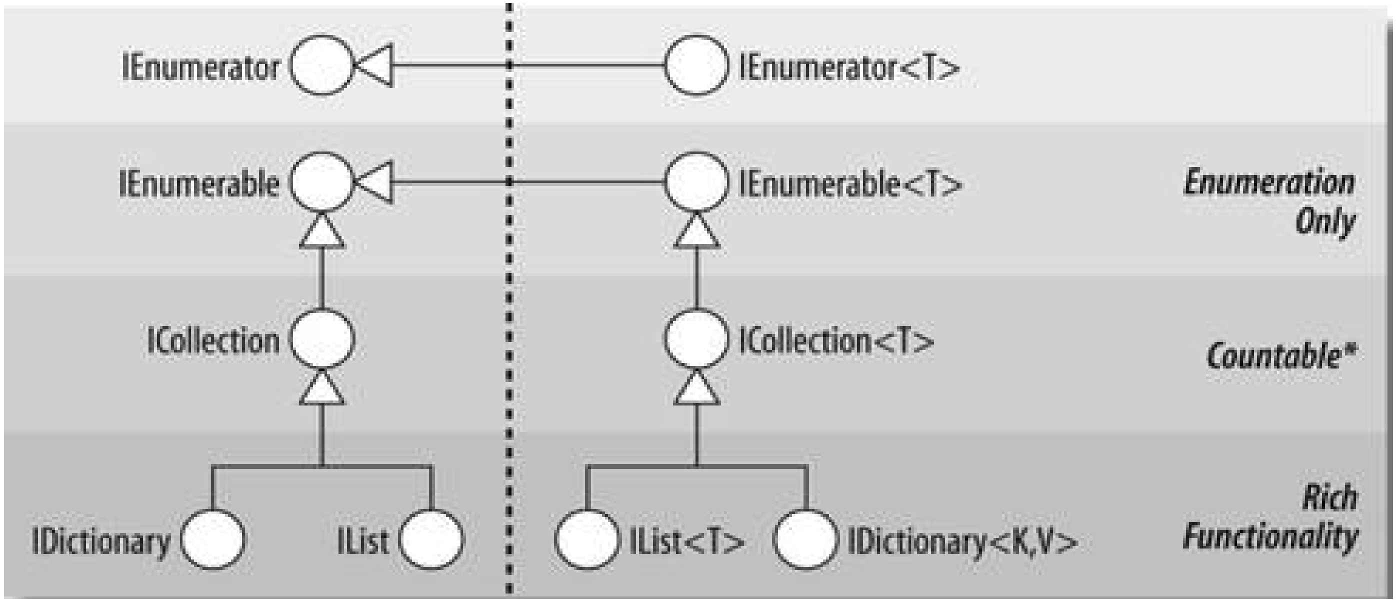
- Differs from C++ templates, which use the source code to create a new type at compile time.

- Hence, when compiling a generic type, the compiler needs to ensure that the code will work for any type.

C# COLLECTIONS

Data Structures in C#

- `System.Collections` for nongeneric collection classes and interfaces.
- `System.Collections.Specialized` for strongly typed nongeneric collection classes.
- `System.Collections.Generic` for generic collection classes and interfaces.
- `System.Collections.ObjectModel` contains proxies and bases for custom collections.



IEnumerator - IEnumerable

```
//System.Collections
public interface IEnumerator {
  bool MoveNext( );
  object Current { get; }
  void Reset( );
}
```

```
//System.Collections
public interface Ienumerable{
  IEnumerator GetEnumerator( );
}
```

```
//System.Collections.Generic
public interface IEnumerator<T> :
 IEnumerator,IDisposable{
  T Current { get; }
}
```

```
//System.Collections.Generic
public interface IEnumerable<T> :
          IEnumerable{
  IEnumerator<T> GetEnumerator( );
}
```

If a class implements the `IEnumerable` interface, then the `foreach` statement can be used on that class.

IEnumerable -foreach

```
class Set : ISet, IEnumerable{
   object[] elems;
   public IEnumerator GetEnumerator(){ …}
   //…
}
…
   Set s=new Set();
   s.add(“ana”);
   s.add(“are”);
   s.add(“mere”);
   foreach(Object o in s){
    Console.WriteLine(“{0} “,o);
   }
```

ICollection/ ICollection<T>

```
public interface ICollection : IEnumerable{
    void CopyTo (Array array, int index);
    int Count {get;}
    bool IsSynchronized {get;}
    object SyncRoot {get;}
}

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

IList / IList<T>

```
public interface IList : ICollection, IEnumerable{
  object this [int index] { get; set }
  bool IsFixedSize { get; }
  bool IsReadOnly  { get; }
  int  Add      (object value);
  void Clear(  );
  bool Contains (object value);
  int  IndexOf  (object value);
  void Insert   (int index, object value);
  void Remove   (object value);
  void RemoveAt (int index);
}
public interface IList<T> : ICollection<T>, IEnumerable<T>,
                  IEnumerable{
  T this [int index] { get; set; }
  int IndexOf (T item);
  void Insert (int index, T item);
 void RemoveAt (int index);
}
```

IDictionary

```
public interface IDictionary : ICollection, IEnumerable{
   IDictionaryEnumerator GetEnumerator(  );
   bool Contains (object key);
   void Add      (object key, object value);
   void Remove   (object key);
   void Clear(  );
   object this [object key] { get; set; }
   bool IsFixedSize         { get; }
   bool IsReadOnly          { get; }
   ICollection Keys         { get; }
   ICollection Values       { get; }
}

public interface IDictionaryEnumerator : IEnumerator
{
   DictionaryEntry Entry { get; }
   object Key { get; }
   object Value { get; }
}
```

The IComparable Interface

This require one method CompareTo which returns

-1 if the first value is less than the second

0 if the values are equal

1 if the first value is greater than the second

```
public interface IComparable {
 int CompareTo(object obj)
}
```

The IComparer Interface

This is similar to IComparable but is designed to be implemented in a class outside the class whose instances are being compared

Compare() works just like CompareTo()

```
public interface IComparer {
 int Compare(object o1, object o2);
}
```

Sorting an ArrayList

To use CompareTo() of IComparable

`ArrayList.Sort()`

To use a custom comparer object

`ArrayList.Sort(IComparer cmp)`

To sort a range

`ArrayList.Sort(int start, int len, IComparer cmp)`

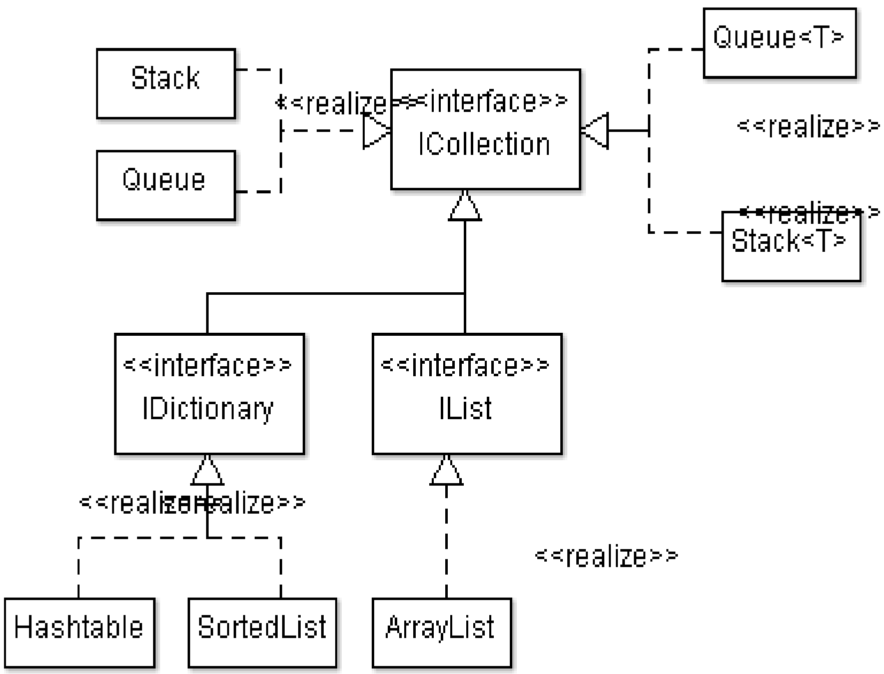
ICloneable Interface

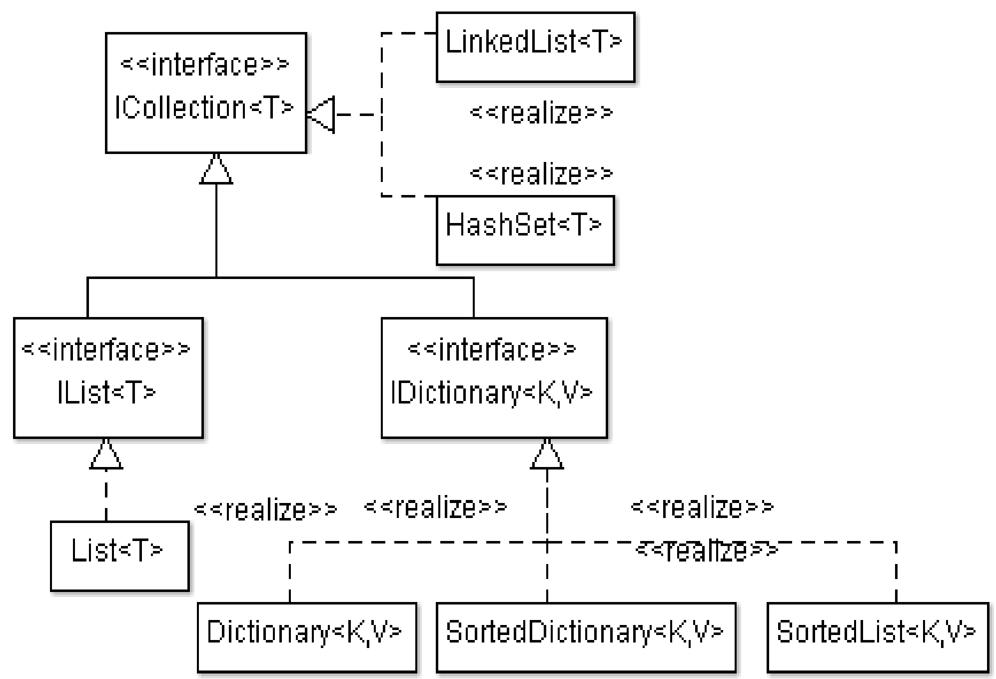
This guarantees that a class can be cloned

The Clone method can be implemented to make a shallow or deep clone

```
public interface ICloneable {
 object Clone();
}
```

Data structures hierarchy





Generic Class	Description
Dictionary<K, V>	Generic unordered dictionary
LinkedList<E>	Generic doubly linked list
List<E>	Generic ArrayList
Queue<E>	Generic queue
SortedDictionary<K, V>	Generic dictionary implemented as a tree so that elements are stored in order of the keys
SortedList<K, E>	Generic binary tree implementation of a list. Can have any type of subscript. More efficient than SortedDictionary in some cases.
Stack<E>	Generic stack

Array Class

The `Array` class is the implicit base class for all single and multidimensional arrays.

It provides a common set of methods to all arrays, regardless of their declaration or underlying element type.

Length and Rank:

```
public int  GetLength      (int dimension);
public int  Length       { get; }
public int Rank { get; }    // Returns number of dimensions in array
```

Searching in a one-dimensional array: `BinarySearch`, `IndexOf`, `LastIndexOf`, etc (static methods, overloaded).

Sorting: `Sort` (overloaded static method).

Reversing elements: `Reverse` (overloaded static method).

Copying: `Copy` (overloaded static method).

IDisposable

Some objects require explicit code to release resources such as open files, locks, operating system handles, and unmanaged objects. This is called dispose, and it is supported through the `IDisposable` interface.

```
public interface IDisposable{
    void Dispose(  );
}
```

IDisposable

- C#'s `using` statement provides a syntactic shortcut for calling `Dispose` on objects that implement `IDisposable`, using a `try / finally` block.

```
using (Font font = new Font("Courier", 12.0f)) {
   //code that uses the object font
}
```

The compiler converts this to:

```
Font font = new Font("Courier", 12.0f);
try{
  // ... Use font
}
finally{
  if (font != null) font.Dispose();
}
```

The `finally` block ensures that the `Dispose` method is called even when an exception is thrown, or the code exits the block early.

IDisposable

- Multiple objects can be used with a `using` statement, but they must be declared inside the `using` statement, or nested:

```
using (Font f3 = new Font("Arial", 10.0f), f4 = new Font("Arial", 9.0f)){
    // Use fonts f3 and f4.
}

using (Font f3 = new Font("Arial", 10.0f))
 using (Font f4 = new Font("Arial", 9.0f)){
    // Use fonts f3 and f4.
}
```

IDisposable

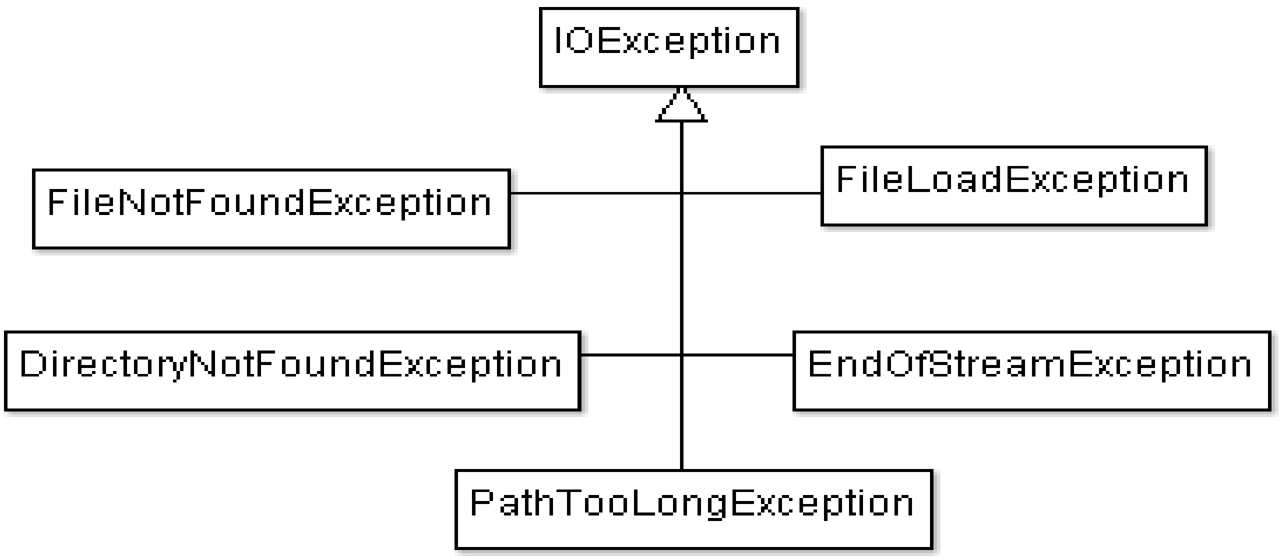
- The Framework follows a set of rules in its disposal logic.

- The rules purpose is to define a consistent protocol to users.
  - Once disposed, an object cannot be reactivated, and calling its methods or properties may throw exceptions or give incorrect results.
  - Calling an object's `Dispose` method repeatedly causes no error.
  - If disposable object `x` contains disposable object `y`, the `Dispose` method of `x` automatically calls the `Dispose` method of `y` - unless instructed otherwise.

C# I/O

I/O

Namespace `System.IO`
Exceptions



Utility I/O Classes

The `System.IO` namespace provides a set of types for performing utility file and directory operations, such as copying, moving, creating directories, and setting file attributes and permissions.

Static classes: `File` and `Directory`

The static methods on File and Directory are convenient for executing a single file or directory operation.

Instance method classes (constructed with a file or directory name): `FileInfo` and `DirectoryInfo`

Static class, `Path`, that provides string manipulation methods for filenames and directory paths.

File / FileInfo

```
public static class File{
   bool Exists (string path);
   void Delete  (string path);
   void Copy    (string sourceFileName, string destFileName);
   void Move    (string sourceFileName, string destFileName);
   void Replace (string source, string destination, String backup);
   FileAttributes GetAttributes (string path);
   void SetAttributes(string path, FileAttributes fileAttributes);
   DateTime GetCreationTime   (string path);
   FileSecurity GetAccessControl (string path);
   void SetAccessControl (string path, FileSecurity fileSecurity);
   //…
}
```

- `FileInfo` offers most of the `File's` static methods in instance form—with some additional properties such as `Extension`, `Length`, `IsReadOnly`

Directory / DirectoryInfo

```
public static class Directory{
    static static bool Exists(string path);
    static void Move(string sourceDirName,string destDirName)
    static string GetCurrentDirectory ();
    static void   SetCurrentDirectory (string path);
    static DirectoryInfo CreateDirectory  (string path);
    static DirectoryInfo GetParent        (string path);
    static string[] GetLogicalDrives();
    static string[] GetFiles(string path);
    static string[] GetDirectories(string path);
    static void Delete(string path)
    //…
}
```

- `DirectoryInfo` exposes instance methods for creating, moving, and enumerating through directories and subdirectories.

Stream Architecture

The .NET stream architecture centers on three concepts: backing stores, decorators, and adapters:

- A *backing store* is the endpoint that makes input and output useful, such as a file or network connection. It is one or both of the following:
  - A *source* from which bytes can be sequentially read.
  - A *destination* to which bytes can be sequentially written.

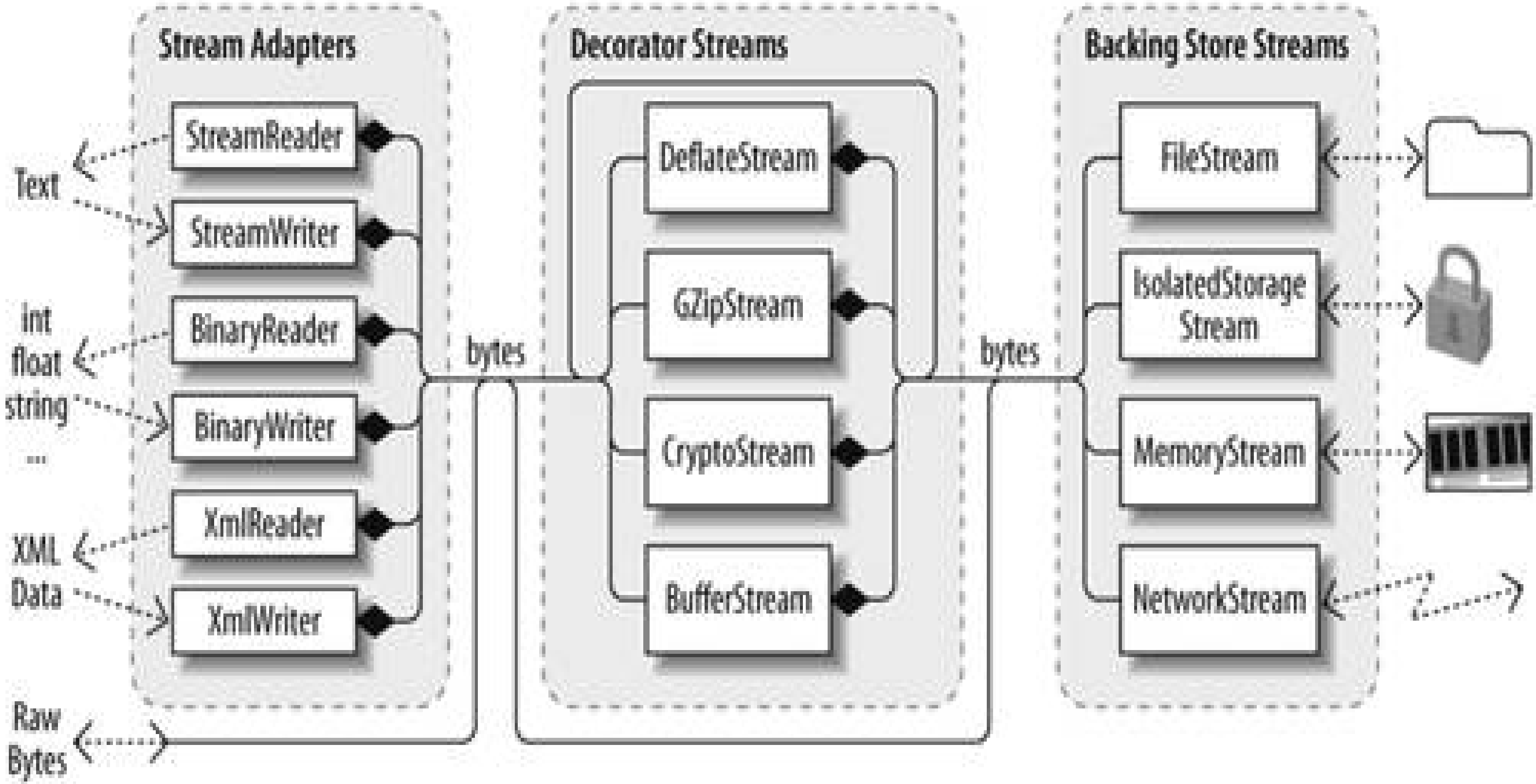
- *Decorator streams* feed off another stream, transforming the data in some way.

- *Adapter* wraps a stream in a class with specialized methods typed to a particular format.

Remark:

An adapter wraps a stream, just as a decorator. Unlike a decorator, however, an adapter is not itself a stream; it typically hides the byte-oriented methods completely.

Stream Architecture



`Stream` class

The abstract `Stream` class is the base for all streams.

It defines methods and properties for three fundamental operations:

- Reading
- Writing
- Seeking
- as well as for administrative tasks such as closing, flushing, and configuring timeouts.

Stream class

Reading

```
public abstract bool CanRead { get; }
public abstract int Read (byte[] buffer, int offset, int count)
public virtual int ReadByte( );
```

Writing

```
public abstract bool CanWrite { get; }
public abstract void Write (byte[] buffer, int offset, int count);
public virtual void WriteByte (byte value);
```

Seeking

```
public abstract bool CanSeek { get; }
public abstract long Position { get; set; }
public abstract void SetLength (long value);
public abstract long Length { get; }
public abstract long Seek (long offset, SeekOrigin origin);
```

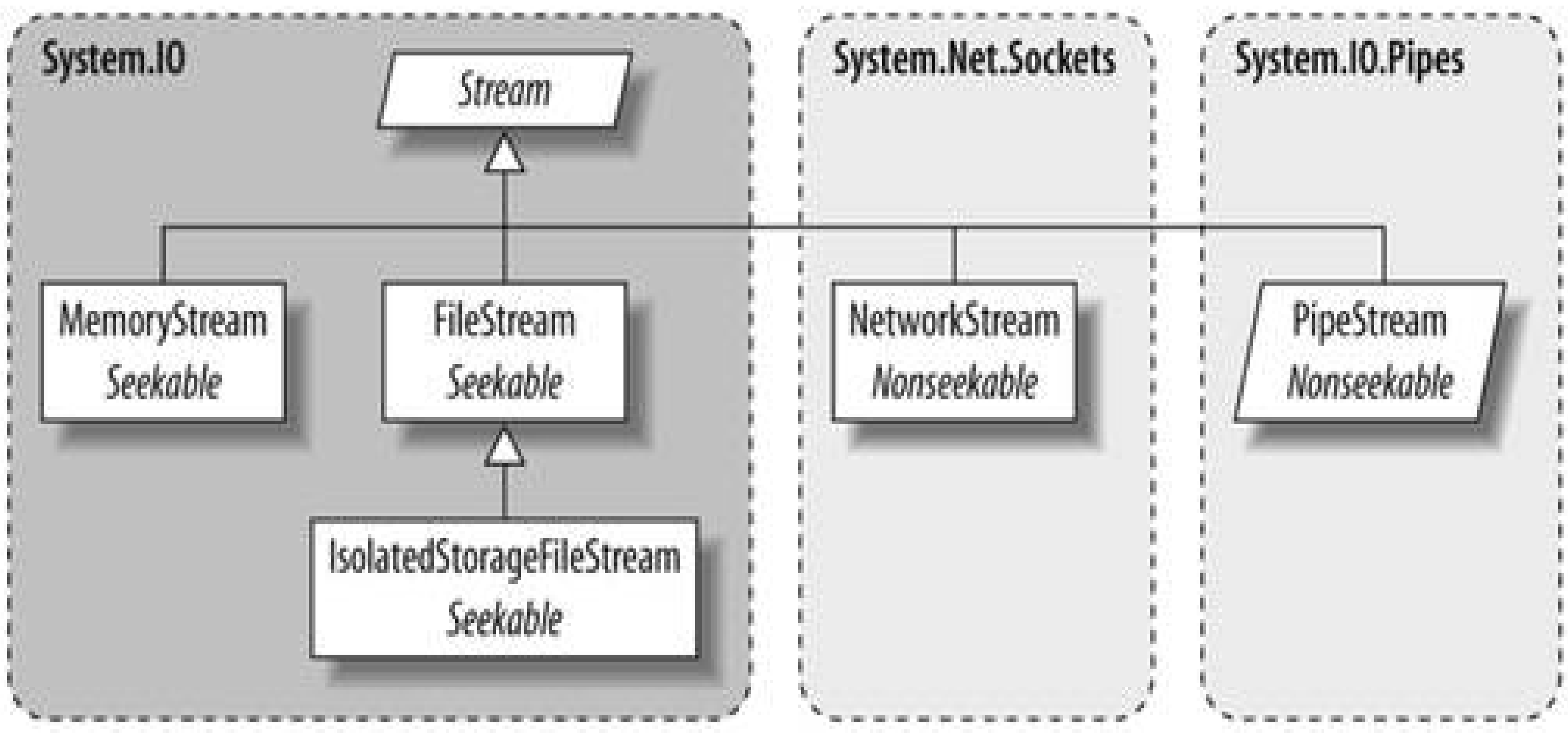
Closing/ flushing

```
public virtual void Close( );
public void Dispose( );
public abstract void Flush( );
```

Timeouts

```
public abstract bool CanTimeout { get; }
public override int ReadTimeout { get; set; }
public override int WriteTimeout { get; set; }
```

Backing Store Streams



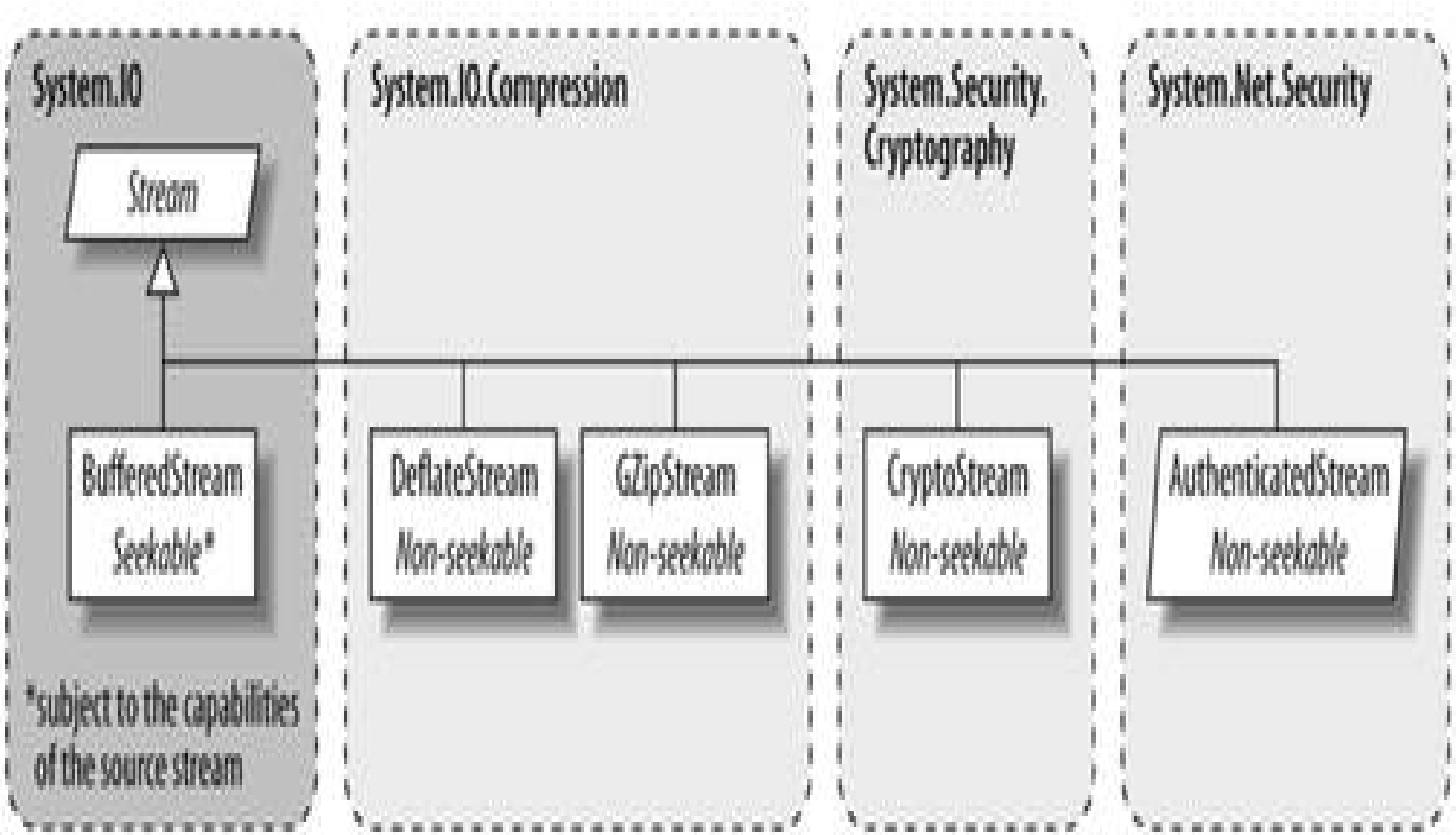
FileStream class

```
public class FileStream : Stream{
   public FileStream(string path, FileMode mode); //overloaded
   public override int Read (byte[] buffer, int offset, int count);
   public override int ReadByte( );
   public override long Seek (long offset, SeekOrigin origin);
   //…
}
public enum SeekOrigin{Begin, Current, End}

public enum FileMode{CreateNew, Create, Open, OpenOrCreate,  Truncate,  Append }
```

FileStream class - example

```
using(FileStream fs=new FileStream("testFileStream.txt", FileMode.Create)) {
  String message = "Ana are mere.";
    //helper class to transform a string in an array of bytes, System.Text
  UnicodeEncoding unienc=new UnicodeEncoding();
  //writing to the file
  fs.Write(unienc.GetBytes(message), 0, unienc.GetByteCount(message));

     //creates an array of bytes for reading the data from the file
   byte[] rbytes=new byte[fs.Length];
     //position the file pointer to the beginning
   fs.Seek(0, SeekOrigin.Begin);
     //reads the data from the file
   fs.Read(rbytes, 0, (int) fs.Length);
     //transforms the bytes in a string
   String newMess=new string(unienc.GetChars(rbytes,0,rbytes.Length));
   Console.WriteLine("Written text {0}, read text {1}", message, newMess);
}
```

Decorator Streams



Stream Adapters

A `Stream` deals only with bytes.

In order to read or write data types such as `strings`, `integers`, or XML elements a stream adapter should be used:

Text adapters (for string and character data):

- `TextReader`, `TextWriter`,
- `StreamReader`, `StreamWriter`,
- `StringReader`, `StringWriter`

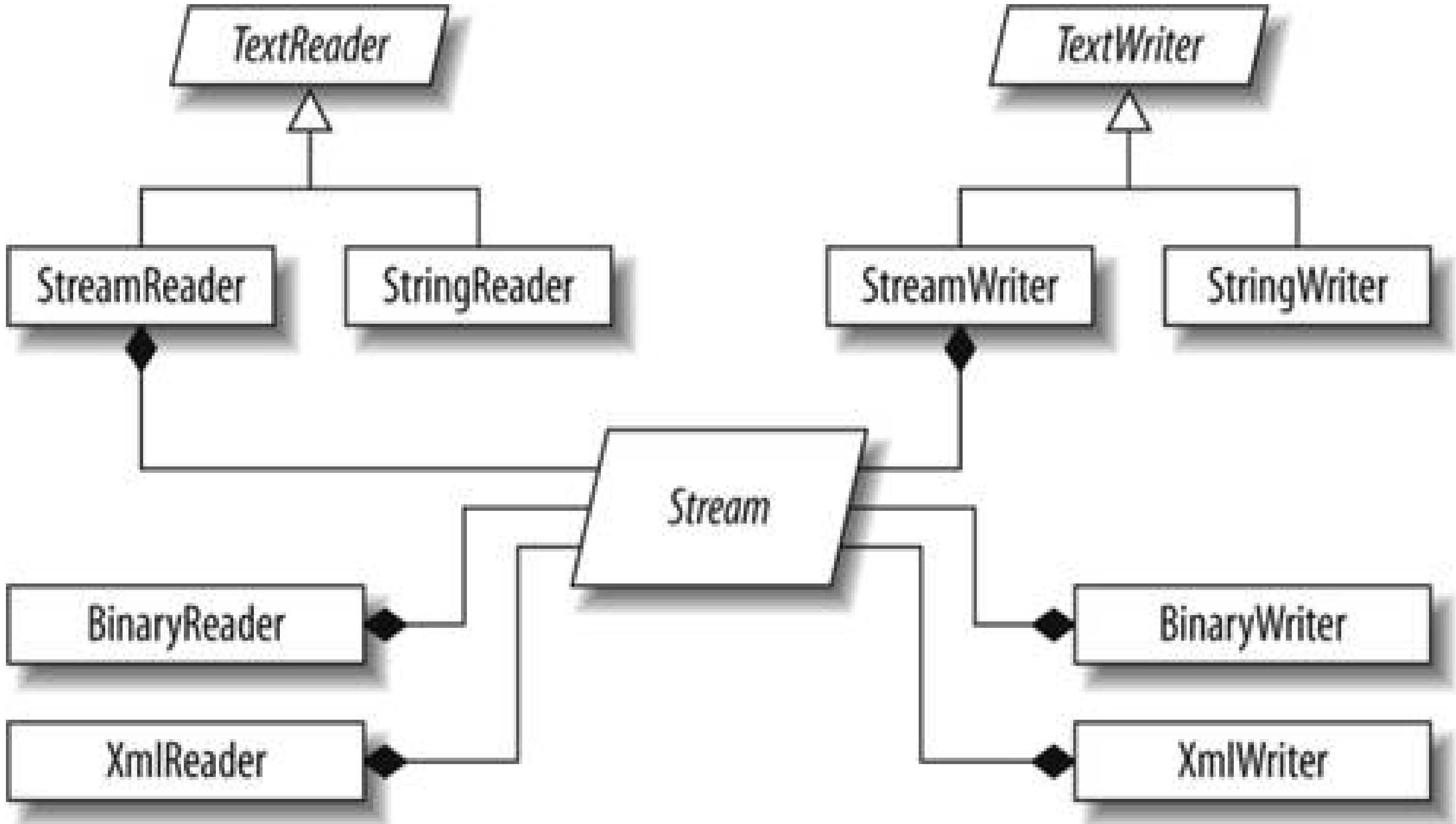
Binary adapters (for primitive types such as `int`, `bool`, `string`, and `float`)

- `BinaryReader`, `BinaryWriter`

XML adapters

- `XmlReader`, `XmlWriter`

Stream Adapters



Text Adapters

- `TextReader` and `TextWriter` are the abstract base classes for adapters that deal exclusively with characters and strings. Each has two general-purpose implementations in the framework:

`StreamReader`/`StreamWriter`: uses a `Stream` for its data store, and translates the stream's bytes into characters or strings

`StringReader`/`StringWriter`: implements `TextReader`/`TextWriter` using in-memory strings

Because text adapters are often associated with files, the `File` class provides the static methods `CreateText`, `AppendText`, and `OpenText` to ease the process of creating file-based text adapters.

Text Adapters

```
//Writing to a text file
using (FileStream fs = File.Create ("test.txt"))
using (TextWriter writer = new StreamWriter (fs))
//or TextWriter writer=File.CreateText("test.txt")
{
  writer.WriteLine ("Ana are mere.");
  writer.WriteLine ("mere");
}
//Reading from a text file
using (FileStream fs = File.OpenRead ("test.txt"))
using (TextReader reader = new StreamReader (fs))
//or TextReader reader=File.OpenText("test.txt")
{
  Console.WriteLine (reader.ReadLine());      // reads line 1
  Console.WriteLine (reader.ReadLine());      // reads line 2
}
```

Binary Adapters

- **`BinaryReader`** and **`BinaryWriter`** read and write native data types: **`bool`**, **`byte`**, **`char`**, **`decimal`**, **`float`**, **`double`**, **`short`**, **`int`**, **`long`**, **`sbyte`**, **`ushort`**, **`uint`**, and **`ulong`**, as well as **`strings`** and **`arrays`** of the primitive data types.

```
using (BinaryWriter w = new BinaryWriter(File.Create("testBinary.txt")))
{
       w.Write("Ana");
       w.Write(23);
       w.Write(12.4);
       w.Flush();
}
using(BinaryReader r = new BinaryReader(File.Open("testBinary.txt",FileMode.Open)))
 {
       String name = r.ReadString();
       int age = r.ReadInt32();
       double d= r.ReadDouble();
       Console.WriteLine("name= {0}, age={1}, d={2}",name,age,d);
  }
```

LINQ

Before LINQ

```
int[] numbers = { 3, 6, 7, 9, 2, 5, 3, 7 };
int i = 0;

// Display numbers larger than 5
while (i < numbers.GetLength(0))
{
if (numbers[i] > 5)
Console.WriteLine(numbers[i]);

++i;
}
```

LINQ

```
int[] numbers = { 3, 6, 7, 9, 2, 5, 3, 7 };
var res = from n in numbers
          where n > 5
          select n;

     foreach (int n in res)
     Console.WriteLine(n);
```

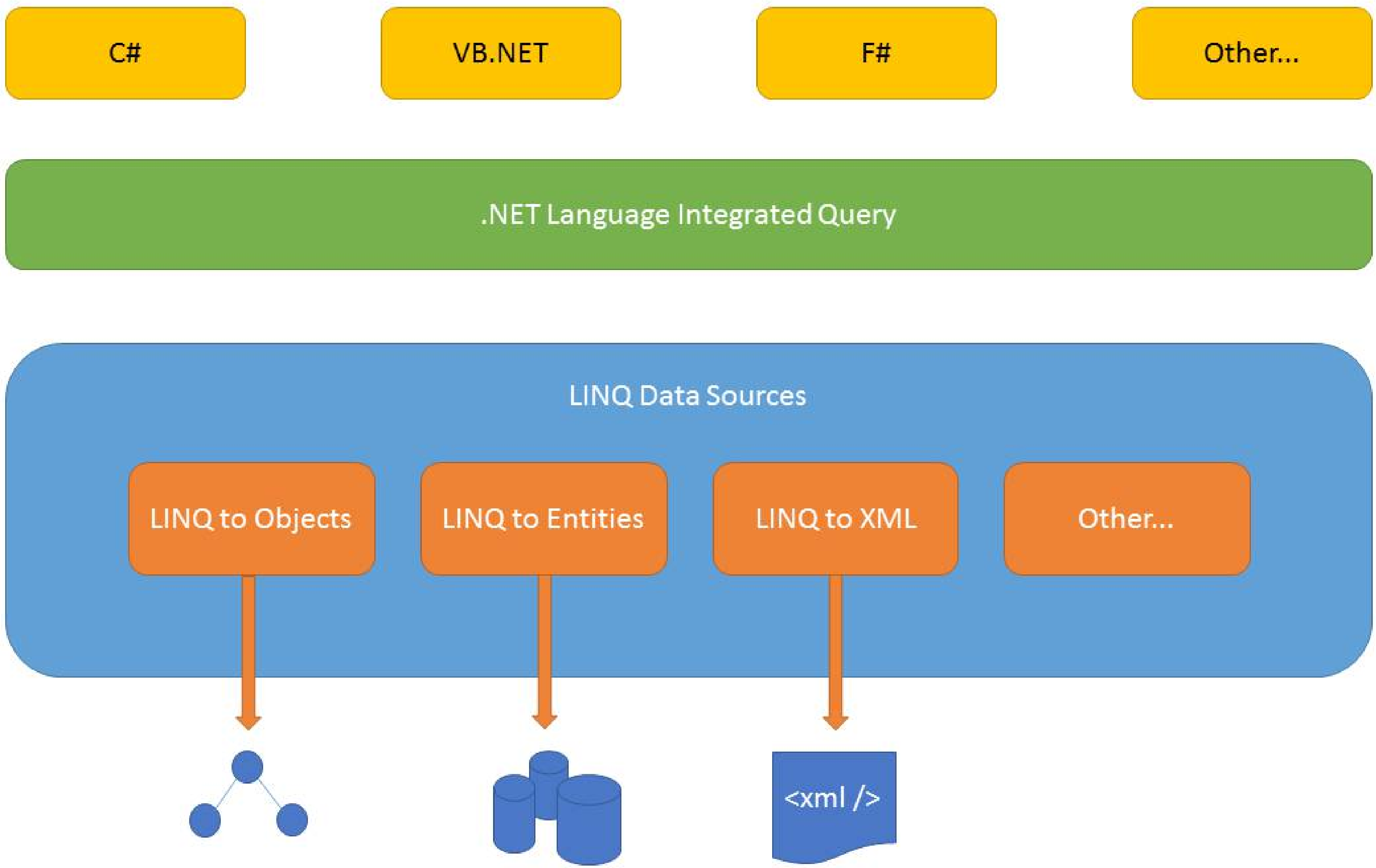
LINQ

- **querying data from various sources, such as arrays, dictionaries, xml and entities created from entity framework.**
- **instead of having to use a different API for each data source, LINQ provides a consistent and uniform programming model to work with all supported data sources.**

Some of the most used LINQ data sources, which are all part the .NET framework, are:

- **LINQ to Objects: for in-memory collections based on *IEnumerable*, such as *Dictionary* and *List*.**
- **LINQ to Entities: for Entity Framework on object context.**
- **LINQ to XML: for in-memory XML documents.**

LINQ


C#
VB.NET
F#
Other...
.NET Language Integrated Query
LINQ Data Sources
LINQ to Objects
LINQ to Entities
LINQ to XML
Other...
<xml />

LINQ

1) with SQL-like syntax called *Query expressions*

```
int[] numbers = { 7, 53, 45, 99 };

    var res = from n in numbers
        where n > 50
        orderby n
        select n.ToString();
```

LINQ

2) a method like approach called *Lambda expressions*

```
int[] numbers = { 7, 53, 45, 99 };

    var res = numbers.Where(n => n > 50)
                     .OrderBy(n => n)
                     .Select(n => n.ToString());
```

LINQ

LINQ queries can execute in two different ways: deferred and immediate.

With deferred execution, the resulting sequence of a LINQ query is not generated until it is required. The following query does not actually execute until Max() is called, and a final result is required:

```
int[] numbers = { 1, 2, 3, 4, 5 };
        var result = numbers.Where(n => n >= 2 && n <= 4);
        Console.WriteLine(result.Max()); // <- query executes at this point

        // Output:
        //4
```

LINQ

Deferred execution makes it useful to combine or extend queries. Have a look at this example, which creates a base query and then extends it into two new separate queries:

```
int[] numbers = { 1, 5, 10, 18, 23};
var baseQuery = from n in numbers select n;
var oddQuery = from b in baseQuery where b % 2 == 1 select b;

Debug.WriteLine("Sum of odd numbers: " + oddQuery.Sum()); // <- query executes at this point

var evenQuery = from b in baseQuery where b % 2 == 0 select b;
Debug.WriteLine("Sum of even numbers: " + evenQuery.Sum()); // <- query executes at this point

// Output:
// Sum of odd numbers: 29
// Sum of even numbers: 28
```