RAWDATA SECTION 2

Troels Andreasen & Henrik Bulskov

ASSIGNMENT 3

LINQ LANGUAGE-INTEGRATED QUERIES

- Language Features
  - Lambda expressions
  - Extensions Methods
  - Anonymous Types
  - Query Expression Syntax
  - Generics
  - yield and var

DELEGATES

- A delegate is an object that knows how to call a method
- A *delegate type* defines the kind of method that *delegate instances* can call, i.e. the signature

```
delegate int Transformer (int x);
```

- Transformer is compatible with any method with an int return type and a single int parameter, e.g.

```
static int Square (int x) { return x * x; }
```

DELEGATES

- A delegate instance literally acts as a delegate for the caller

```
Transformer t = Square;
```

- Inworking the delegate, calls the target method

```
t.Invoke(3)        or    t(3)
```

- Delegate instances have multicast capability
- Using the += and -= to add or remove methods

```
SomeDelegate d = SomeMethod1;
d += SomeMethod2;
```

- Invoking d will now call both SomeMethod1 and SomeMethod2, in the order they are added.

GENERIC DELEGATES

- A delegate type may contain generic type parameters

```
public delegate T Transformer<T> (T arg);
```

- Func and Action Delegates are defined in the System namespace
- Func

```
delegate TResult Func <out TResult>                 ();
delegate TResult Func <in T, out TResult>           (T arg);
delegate TResult Func <in T1, in T2, out TResult>   (T1 arg1, T2 arg2);
... and so on, up to T16
```

- Action

```
delegate void Action                  ();
delegate void Action <in T>           (T arg);
delegate void Action <in T1, in T2>   (T1 arg1, T2 arg2);
... and so on, up to T16
```

LAMBDA EXPRESSIONS

- A lambda expression is an unnamed method written in place of a delegate instance

```
delegate int Transformer (int x);

Transformer sqr = x => x * x;
Console.WriteLine (sqr(3));    // 9
```

- A lambda e: *(parameters) => expression-or-statement-block*

```
x => x * x;

x => { return x * x; };
```

- Lambda expressions are used most commonly with the Func and Action delegates

```
Func<int,int> sqr = x => x * x;
```

LAMBDA EXPRESSIONS PARAMETER TYPES

- The compiler can usually infer the type of lambda parameters contextually

```
Func<int,int> sqr = x => x * x;
```

- Otherwise explicitly specify the types

```
Func<int,int> sqr = (int x) => x * x;
```

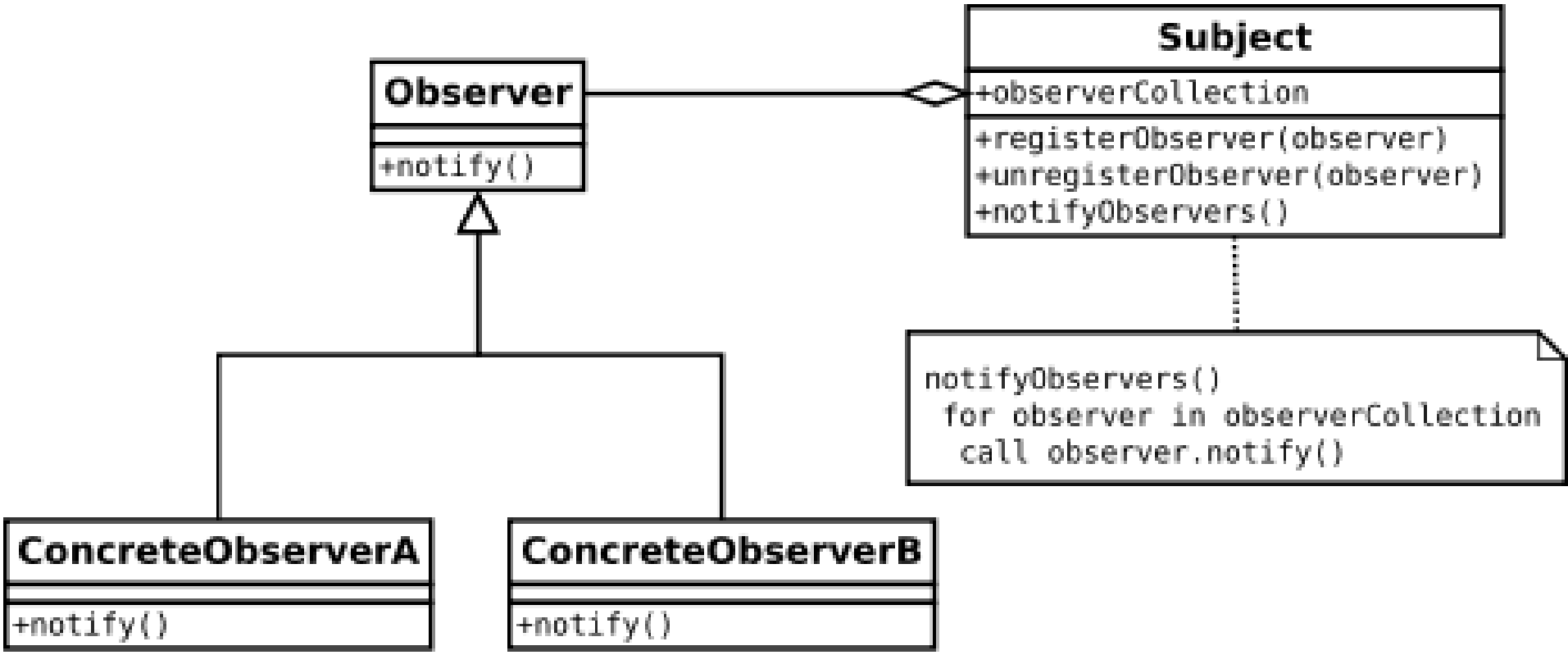
OUTER VARIABLES - CLOSURE

- Lambda expression can reference the local variables and parameters of the method in which it's defined

```
int factor = 2;
Func<int, int> multiplier = n => n * factor;
Console.WriteLine (multiplier (3));          // 6
```

- Outer variables referenced by a lambda expression are called *captured variables*.
- A lambda expression that captures variables is called a *closure*
- When you capture the iteration variable of a for loop, C# treats that variable as though it was declared *outside* the loop!!!

OBSERVER PATTERN



https://en.wikipedia.org/wiki/Observer_pattern

EVENTS

```
// Delegate definition
public delegate void PriceChangedHandler (decimal oldPrice,
                                          decimal newPrice);
public class Broadcaster
{
  // Event declaration
  public event PriceChangedHandler PriceChanged;
}
```

- Subscription to an event if done by use of the += operator
- If any subscribers

  we can invoke them with:

```
if (PriceChanged != null)
  PriceChanged (oldPrice, price);
```

- Standard Event Pattern

```
System.EventArgs
```

- Rules for event delegate:
  - It must have a void return type
  - It must accept two arguments: the first of type object, and the second a subclass of EventArgs
  - Its name must end with *EventHandler*

```
public delegate void EventHandler<TEventArgs>
  (object source, TEventArgs e) where TEventArgs : EventArgs;
```

ENUMERATION AND ITERATORS

- An enumerator is a read-only, forward-only cursor over a sequence of values.
- An enumerator is an object that implements either of the following interfaces:

```
System.Collections.IEnumerator
System.Collections.Generic.IEnumerator<T>
```

- Technically, any object that has a method named MoveNext and a property called Current is treated as an enumerator
- The foreach statement iterates over *enumerable* objects

ITERATOR

- An iterator is a method, property, or indexer that contains one or more yield statements, and return of the four interfaces

```
System.Collections.IEnumerable                  System.Collections.IEnumerator
System.Collections.Generic.IEnumerable<T>       System.Collections.Generic.IEnumerator<T>
```

- yield

```
static IEnumerable<string> Foo()
{
  yield return "One";
  yield return "Two";
  yield return "Three";
}
```

Extend any type with additional methods

LINQ provides extension methods on IEnumerable<T>

Connect these extension methods together into “pipelines”

EXTENSIONS METHODS

EXTENSIONS METHODS - EXAMPLE

```
0 references | 0 changes | 0 authors, 0 changes
static class MyExt
{
    1 reference | 0 changes | 0 authors, 0 changes
    public static string Reverse(this string s)
    {
        return string.Join("", s.ToArray().Reverse());
    }
}
0 references | 0 changes | 0 authors, 0 changes
class Program
{
    0 references | 0 changes | 0 authors, 0 changes
    static void Main(string[] args)
    {
        Console.WriteLine("Hello".Reverse());
    }
}
```

ANONYMOUS TYPES

```
var a = new {Name = "Peter", Age = 23};

var b = new {First = 7, Last = 54, Time = DateTime.Now};

var title = "Some title";
var volume = 7;


var c = new {title, volume};

Console.WriteLine("c = " + c);
```

```
c = { title = Some title, volume = 7 }

Press any key to continue . . .
```

```
class Node<K,V>{
  K Key{ get; set;}
  V Value{ get; set;}
  Node<K,V> parent, right, left;

  Node(K key,V value){...}
}
```

Any type but not void

```
class Tree<K,V>{
  Node<K,V> root;

  Tree(){...}

  Add(K key, V value){...}

  Remove(K key){...}

  Find(K key){...}
}
```

GENERIC CLASSES

CONSTRAINTS ON TYPE PARAMETERS

Can instances of K be compared ?

Does V have a default constructor?

```
class Tree<K,V>{
  Node<K,V> root;

  Tree(){...}

  Add(K key, V value){...}

  Remove(K key){...}

  Find(K key){...}
}
```

CONSTRAINTS ON TYPE PARAMETERS

Forces K to implement the IComparable<K> interface

```
class Tree<K,V>
      where K : IComparable<K>
{
   Node<K,V> root;

   Tree(){...}

   Add(K key, V value){...}

   Remove(K key){...}

   Find(K key){...}
}
```

CONSTRAINTS ON TYPE PARAMETERS

```
class Tree<K,V>
      where K : IComparable<K>
      where V : new()
{
  Node<K,V> root;

  Tree(){...}

  Add(K key, V value){...}

  Remove(K key){...}

  Find(K key){...}
}
```

Forces V to have an argumentless constructor

CONSTRAINTS ON TYPE PARAMETERS

```
class Tree<K,V>
      where K : IComparable<K>
      where V : new() , class
{
  Node<K,V> root;

  Tree(){...}

  Add(K key, V value){...}

  Remove(K key){...}

  Find(K key){...}
}
```

Forces V to have an argumentless constructor AND be a reference type

CONSTRAINTS ON TYPE PARAMETERS

```
class Tree<K,V>
      where K : IComparable<K>
      where V : struct
{
  Node<K,V> root;

  Tree(){...}

  Add(K key, V value){...}

  Remove(K key){...}

  Find(K key){...}
}
```

Forces V be a value type

```
where T : base-class    // Base-class constraint
where T : interface     // Interface constraint
where T : class         // Reference-type constraint
where T : struct        // Value-type constraint (excludes Nullable types)
where T : new()         // Parameterless constructor constraint
where U : T             // Naked type constraint
```

GENERIC CONSTRAINTS

USE OF TYPE PARAMETERS

- Use it as type of fields, variables, properties, method parameters and return types
- Use it to create arrays e.g. `new T[10]`
- Call `default(T)` to get the appropriate default value
- Create a new instance with `new T()` if the new() constraint is specified
- Use methods of the interfaces or base classes in the constraint specification.
- CANNOT call static methods

GENERIC METHODS

```
public static class ListExtensions
{
  public static void Scramble<T>(this List<T> array)
  {
    Random rand = new Random();
    int j = 0;
    for(int i=0; i<array.Count; i++)
    {
      j = rand.Next(array.Count);
      T tmp = array[j];
      array[j] = array[i];
      array[i] = tmp;
    }
  }
}
```

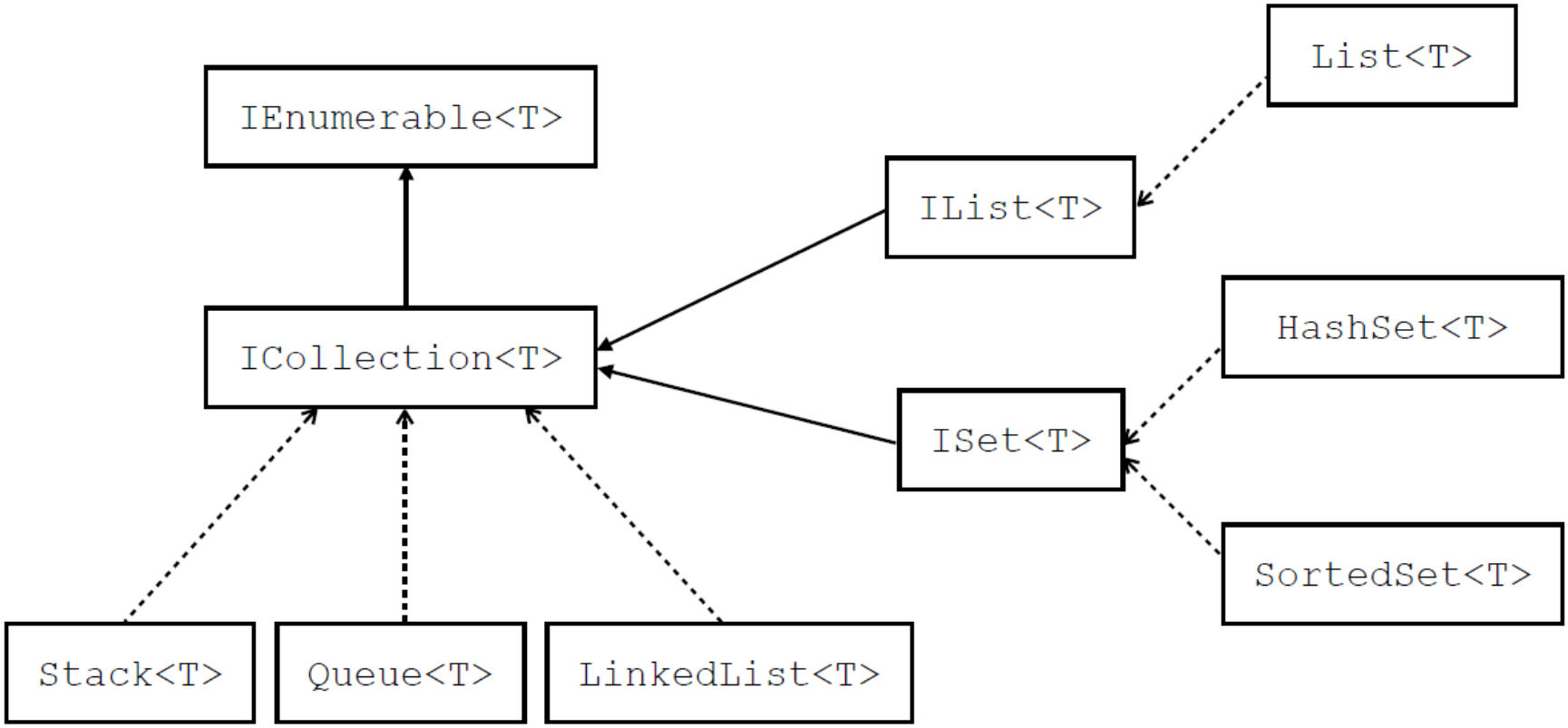
```
List<int> list = new List<int>{1,2,3,4};
list.Scramble();// list={3,2,4,1}
```

GENERIC METHODS

```
public static class ListExtensions
{
   public static void Scramble<T>(this List<T> array){...}

   public static void StableSort<T>(this List<T> array)
                                        where T : IComparable<T>
   {
      ...
   }
}
```

Overridden generic methods or methods implementing generic interfaces must have the same parameter constraints.

COLLECTIONS

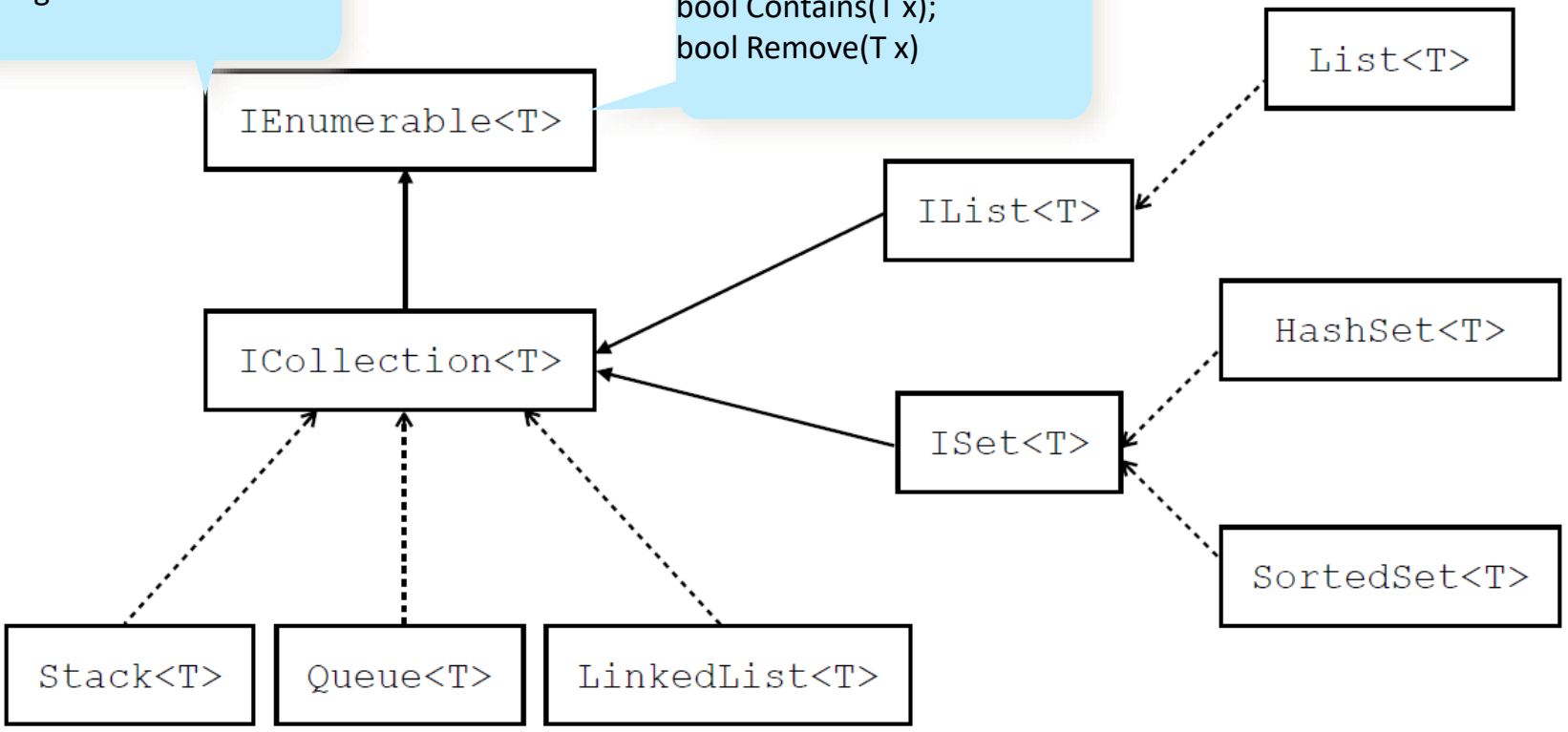


COLLECTIONS

E.g. Allows enumerations within foreach loop through IEnumerator<T>

Some Shared Methods

int Count;
void Add(T x);
void Clear();
bool Contains(T x);
bool Remove(T x)



```
List<int> list = new List<int>();
list.Add(3);
list.Add(5);
list.Add(6);
Console.Out.WriteLine(list[2]);// writes: 6
```

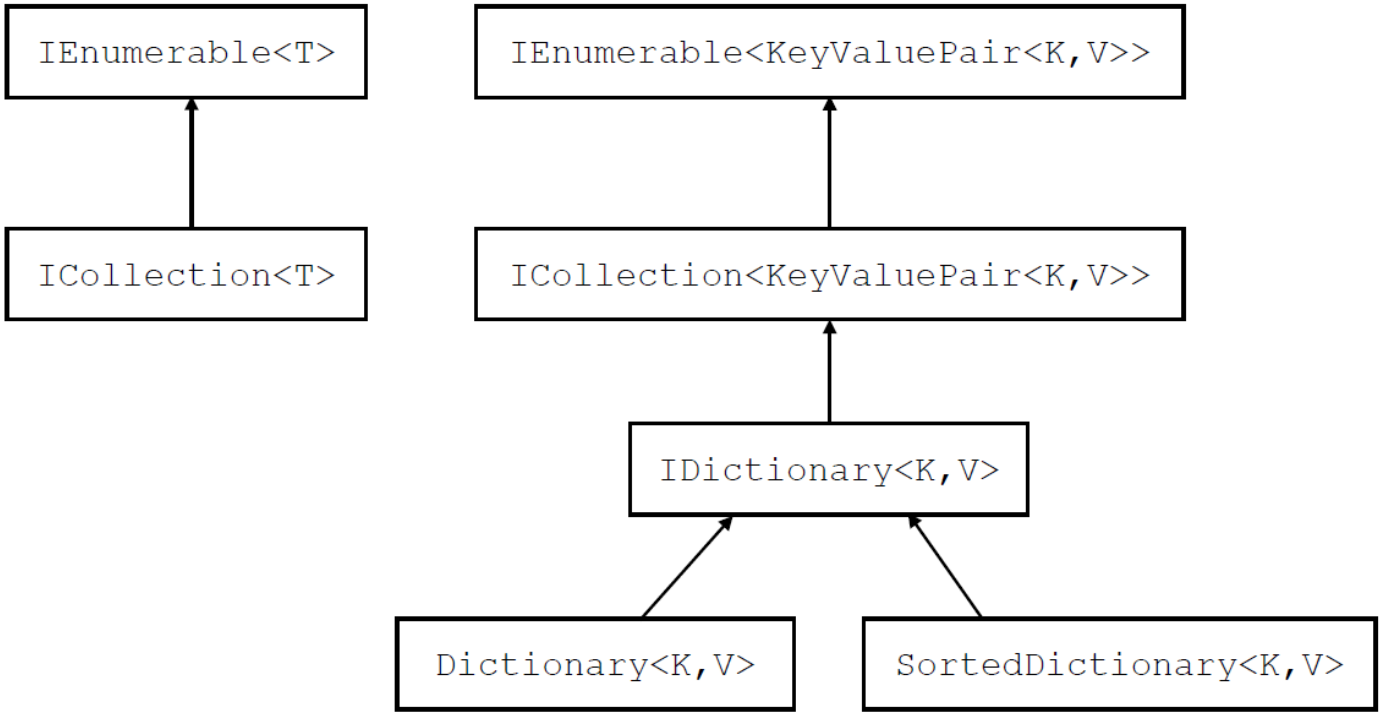
```
List<int> list = new List<int>{3,4,6};
Console.Out.WriteLine(list[2]);// writes: 6
```

LIST<T>

LIST<T>

```
struct Contact
{
  public int Number;
  public string Name;
  public Contact(int number, string name){...}
  public override string ToString(){...}
}
```

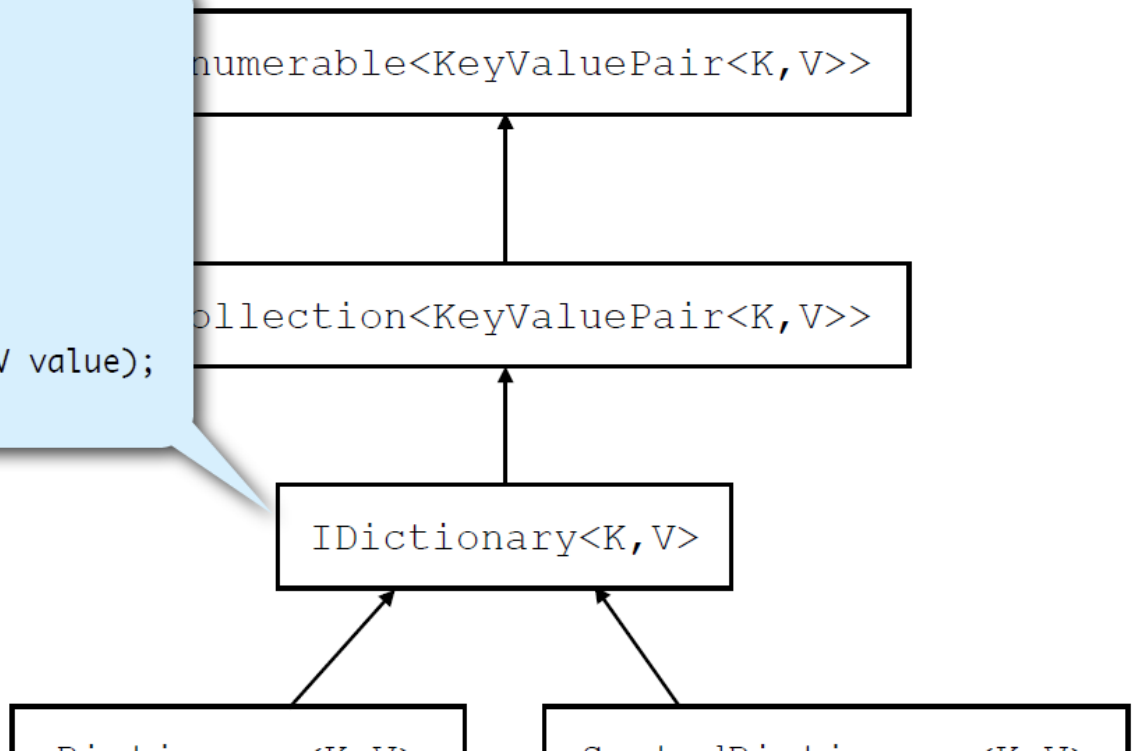
```
List<Contact> contacts = new List<Contact>{
  new Contact(123,"Tom"),
  new Contact(345,"Fred")
};
foreach(Contact c in contacts)
{
  Console.Out.WriteLine(c);
}
```



KEYVALUE COLLECTIONS

Some Shared Methods

```
ICollection<K>  Keys;
ICollection<V>  Values;
V this[K key];
void Add(K key,V value);
bool Remove(K key);
bool ContainsKey(K key);
bool TryGetValue(K key, out V value);
```

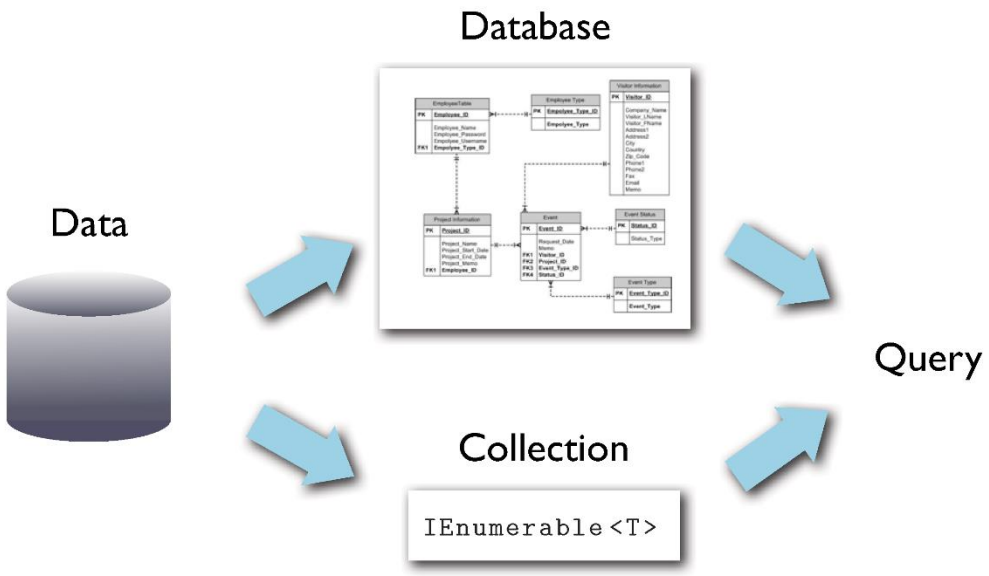


KEYVALUE COLLECTIONS

```
Dictionary<string, int> Variable = new Dictionary<string,int>();

Variable["x_1"] = 30;
Variable["x_2"] = 60;

Console.Out.WriteLine(Variable["x_1"]+Variable["x_2"]);
```

DICTONARY<K,V>
SORTEDDICTIONARY<K,V>

LINQ



LINQ

```
IEnumerable<string> query = names
   .Where   (n => n.Contains ("a"))
   .OrderBy (n => n.Length)
   .Select  (n => n.ToUpper());
```

FLUENT SYNTAX

Chaining Query Operators

```
IEnumerable<string> query =
  from    n in names
  where   n.Contains ("a")     // Filter elements
  orderby n.Length             // Sort elements
  select  n.ToUpper();         // Translate each element (project)
```

QUERY EXPRESSIONS

C# provides a syntactic shortcut for writing LINQ queries, called query expressions

DEFERRED EXECUTION

- An important feature of most query operators is that they execute not when constructed, but when enumerated

```
var numbers = new List<int>();
numbers.Add (1);

IEnumerable<int> query = numbers.Select (n => n * 10);    // Build query

numbers.Add (2);                          // Sneak in an extra element

foreach (int n in query)
  Console.Write (n + "|");                // 10|20|
```

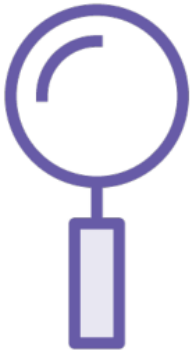
```
string[] musos =
  { "David Gilmour", "Roger Waters", "Rick Wright", "Nick Mason" };

IEnumerable<string> query = musos.OrderBy (m => m.Split().Last());
```

SUBQUERIES

A subquery is a query contained within another query's lambda expression

STRATEGIES

- Composition Strategies
  - Progressive Query Building
- Projection Strategies
  - Anonymous Types



Most Concise
Solves the problem in the fewest lines of code

Most Readable
More code, but easier to understand what's going on

Fastest
More complicated but produces results quickly

CLEAN CODE*

What Are We Aiming For?

* Book of Robert C. Martin – Read It!

FILTERING

Method	Description	SQL equivalents
`Where`	Returns a subset of elements that satisfy a given condition	`WHERE`
`Take`	Returns the first `count` elements and discards the rest	`WHERE ROW_NUMBER()...` *or* `TOP n` subquery
`Skip`	Ignores the first `count` elements and returns the rest	`WHERE ROW_NUMBER()...` *or* `NOT IN (SELECT TOP n...)`
`TakeWhile`	Emits elements from the input sequence until the predicate is false	Exception thrown
`SkipWhile`	Ignores elements from the input sequence until the predicate is false, and then emits the rest	Exception thrown
`Distinct`	Returns a sequence that excludes duplicates	`SELECT DISTINCT...`

PROJECTING

Method	Description	SQL equivalents
`Select`	Transforms each input element with the given lambda expression	`SELECT`
`SelectMany`	Transforms each input element, and then flattens and concatenates the resultant subsequences	`INNER JOIN`, `LEFT OUTER JOIN`, `CROSS JOIN`

JOINING

Method	Description	SQL equivalents
`Join`	Applies a lookup strategy to match elements from two collections, emitting a flat result set	`INNER JOIN`
`GroupJoin`	As above, but emits a *hierarchical* result set	`INNER JOIN`, `LEFT OUTER JOIN`
`Zip`	Enumerates two sequences in step (like a zipper), applying a function over each element pair	

ORDERING

Method	Description	SQL equivalents
`OrderBy, ThenBy`	Sorts a sequence in ascending order	ORDER BY ...
`OrderByDescending, ThenByDescending`	Sorts a sequence in descending order	ORDER BY ... DESC
`Reverse`	Returns a sequence in reverse order	Exception thrown

GROUPING

Method	Description	SQL equivalents
GroupBy	Groups a sequence into subsequences	GROUP BY

SET OPERATORS

Method	Description	SQL equivalents
`Concat`	Returns a concatenation of elements in each of the two sequences	`UNION ALL`
`Union`	Returns a concatenation of elements in each of the two sequences, excluding duplicates	`UNION`
`Intersect`	Returns elements present in both sequences	`WHERE ... IN (...)`

`Except`	Returns elements present in the first, but not the second sequence	`EXCEPT`
		or
		`WHERE ... NOT IN (...)`

CONVERSION METHODS

Method	Description
`OfType`	Converts `IEnumerable` to `IEnumerable<T>`, discarding wrongly typed elements
`Cast`	Converts `IEnumerable` to `IEnumerable<T>`, throwing an exception if there are any wrongly typed elements
`ToArray`	Converts `IEnumerable<T>` to `T[]`
`ToList`	Converts `IEnumerable<T>` to `List<T>`
`ToDictionary`	Converts `IEnumerable<T>` to `Dictionary<TKey,TValue>`
`ToLookup`	Converts `IEnumerable<T>` to `ILookup<TKey,TElement>`
`AsEnumerable`	Downcasts to `IEnumerable<T>`
`AsQueryable`	Casts or converts to `IQueryable<T>`

ELEMENT OPERATORS

Method	Description	SQL equivalents
`First, FirstOrDefault`	Returns the first element in the sequence, optionally satisfying a predicate	`SELECT TOP 1 ... ORDER BY ...`
`Last, LastOrDefault`	Returns the last element in the sequence, optionally satisfying a predicate	`SELECT TOP 1 ... ORDER BY ... DESC`
`Single, SingleOrDefault`	Equivalent to `First/FirstOrDefault`, but throws an exception if there is more than one match	
`ElementAt, ElementAtOrDefault`	Returns the element at the specified position	Exception thrown
`DefaultIfEmpty`	Returns a single-element sequence whose value is `default(TSource)` if the sequence has no elements	`OUTER JOIN`

AGGREGATION METHODS

Method	Description	SQL equivalents
`Count, LongCount`	Returns the number of elements in the input sequence, optionally satisfying a predicate	`COUNT (...)`
`Min, Max`	Returns the smallest or largest element in the sequence	`MIN (...), MAX (...)`
`Sum, Average`	Calculates a numeric sum or average over elements in the sequence	`SUM (...), AVG (...)`
`Aggregate`	Performs a custom aggregation	Exception thrown

QUANTIFIERS

Method	Description	SQL equivalents
`Contains`	Returns `true` if the input sequence contains the given element	`WHERE ... IN (...)`
`Any`	Returns `true` if any elements satisfy the given predicate	`WHERE ... IN (...)`
`All`	Returns `true` if all elements satisfy the given predicate	`WHERE (...)`
`SequenceEqual`	Returns `true` if the second sequence has identical elements to the input sequence	

GENERATION METHODS

Method	Description
`Empty`	Creates an empty sequence
`Repeat`	Creates a sequence of repeating elements
`Range`	Creates a sequence of integers