

# C# Language & .NET Platform

## 11<sup>th</sup> Lecture:

# BCL Classes & Interfaces

<http://d3s.mff.cuni.cz/~jezek>

Department of  
Distributed and  
Dependable  
Systems



*Pavel Ježek*

*pavel.jezek@d3s.mff.cuni.cz*



CHARLES UNIVERSITY IN PRAGUE  
faculty of mathematics and physics


Some of the slides are based on University of Linz .NET presentations.  
© University of Linz, Institute for System Software, 2004  
published under the Microsoft Curriculum License  
([http://www.msdn.com/curriculum/license\\_curriculum.aspx](http://www.msdn.com/curriculum/license_curriculum.aspx))

# IEnumerable and IEnumerator (1)

- Anything which is enumerable is represented by interface IEnumerable

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

- IEnumerator realizes an iterator



```
interface IEnumerator {  
    object Current {get;}  
    bool MoveNext();  
    void Reset();  
}
```

- Also generic versions IEnumerable<T> and IEnumerator<T>

# Collection Classes

- .NET 1.0, 1.1 based on System.Object

**System.Collections** namespace

- .NET 2.0, 3.0, 3.5, 4.0 generic classes

**System.Collections.Generic** namespace

# Interface ICollection : IEnumerable

## Interface ICollection<T> : IEnumerable, IEnumerable<T>

- Basic interface for collections:

`int Count { get; }`

- number of elements

`bool IsSynchronized {get;}`

- collection synchronised?

`object SyncRoot {get;}`

- returns object for synchronisation

`void CopyTo(Array a, int index);`

- copies the elements into array (starting at position index)

- New in ICollection<T>:

`void Add(T item);`

`bool Remove(T item);`

`void Clear();`

`bool Contains(T item);`

# Interface IList : ICollection, IEnumerable

## Interface IList<T> : ICollection<T>, IEnumerable, IEnumerable<T>

- Interface for object collections with a defined order

```
interface IList {  
    object this [ int index ] {get; set;}  
  
    int Add(object value);  
  
    void Insert(int index,object value);  
  
    void Remove(object value);  
  
    void RemoveAt(int index);  
  
    void Clear();  
    bool Contains(object value);  
    ...  
}
```

# Interface IDictionary

## Interface IDictionary<T, U>

```
interface IDictionary : ICollection, IEnumerable {
```

```
    ICollection Keys {get;};
```

```
    ICollection Values {get;};
```

```
    object this[object key] {get; set;}
```

```
    void Add(object key, object value);
```

```
    void Remove(object key);
```

```
    bool Contains(object key);
```

```
    IDictionaryEnumerator GetEnumerator();
```

```
    ...
```

```
}
```

```
interface IDictionary<T, U> : IDictionary<TKey, TValue>,
    ICollection<KeyValuePair<TKey, TValue>>,
    IEnumerable<KeyValuePair<TKey, TValue>>, ... {
```

```
    ...
```

```
}
```

```
public struct DictionaryEntry {
    public DictionaryEntry
        (object key, object value);

    public object Key {get;};
    public object Value {get;};
}
```

# Namespace *System.Collections.Generic*

## *Classes*

List<T>

SortedList<T, U>

Dictionary<T, U>

SortedList<T, U>

Stack<T>

Queue<T>

LinkedList<T>

HashSet<T>

SortedSet<T>

implemented as a reallocated T[] (corresponds to *ArrayList*)  
implemented as a pair of realloc. T[], U[] (corr. to *SortedList*)  
implemented as a hash table in an array (corr. to *Hashtable*)  
implemented as a red-black tree  
implemented in a reallocated T[] (corresponds to *Stack*)  
implemented as a circular queue in r. T[] (corresponds to *Queue*)  
doubly linked list of *LinkedListNode<T>* (***new in .NET 2.0***)  
implemented as a hash table in an array + provides quick set operations (***new in .NET 3.5 + implements ISet<T> in 4.0***)  
implemented as a red-black tree (***new in .NET 4.0***)

## *Interfaces*

ICollection<T>

IList<T>

IDictionary<T, U>

ISet<T>

IEnumerable<T>

IEnumerator<T>

IComparable<T>

IComparer<T>

IEquatable<T>

IEqualityComparer<T>

(***new in .NET 4.0***)

# IDisposable

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

```
public class ObjectDisposedException : InvalidOperationException {}
```

# Class System.Object



Topmost base class of all other classes

```
class Object {  
    protected object MemberwiseClone() {...}  
    public Type GetType() {...}  
    public virtual bool Equals (object o) {...}  
    public virtual string ToString() {...}  
    public virtual int GetHashCode() {...}  
    public static bool ReferenceEquals(object objA, object objB);  
}
```

# IEqualityComparer<T>



```
public interface IEqualityComparer<T> {  
    int GetHashCode(T obj);  
    bool Equals(T x, T y);  
}
```

Used by Dictionary<T> and HashSet<T> collections.

EqualityComparer<T>.Default: a default implementation of a IEqualityComparer<T> for type T

# Class System.Object



Topmost base class of all other classes

```
class Object {  
    protected object MemberwiseClone() {...}  
    public Type GetType() {...}  
    public virtual bool Equals (object o) {...}  
    public virtual string ToString() {...}  
    public virtual int GetHashCode() {...}  
    public static bool ReferenceEquals(object objA, object objB);  
}
```

# Sorting: IComparable and IComparer

- IComparable is interface for types with order

```
public interface IComparable {  
    int CompareTo(object obj); // <0 if this < obj, 0 if this == obj, >0 if this > obj  
}
```

```
public interface IComparable<T> {  
    int CompareTo(T obj); // <0 if this < obj, 0 if this == obj, >0 if this > obj  
}
```

classes implementing IComparable are

- values types like Int32, Double, DateTime, ...
- class Enum as base class of all enumeration types
- class String

- IComparer is interface for the realization of compare operators

```
public interface IComparer {  
    int Compare(object x, object y); // <0 if x < y, 0 if x == y, >0 if x > y  
}
```

```
public interface IComparer<T> {  
    int Compare(T x, T y); // <0 if x < y, 0 if x == y, >0 if x > y  
}
```

# Custom IComparer Implementation

- Creation of table of strings:

```
string[][] Table = {  
    new string[] {"John", "Dow", "programmer"},  
    new string[] {"Bob", "Smith", "agent"},  
    new string[] {"Jane", "Dow", "assistant"},  
    new string[] {"Jack", "Sparrow", "manager"}  
};
```

- Printing the table:

```
foreach (string[] Row in Table) {  
    Console.WriteLine(String.Join(", ", Row));  
}
```

# Custom IComparer Implementation (2)

- Comparer for single table (array) column:

```
class ArrayComparer<T> : IComparer<T[]> where T : IComparable<T> {
    private int m_Index;

    public ArrayComparer(int Index) {
        m_Index = Index;
    }

    public int Compare(T[] x, T[] y) {
        return x[m_Index].CompareTo(y[m_Index]);
    }
}
```

- Printing the table:

```
Array.Sort(Employees, new ArrayComparer<string>(2));

foreach (string[] Row in Employees) {
    Console.WriteLine(String.Join(", ", Row));
}
```

```
Bob, Smith, agent
Jane, Dow, assistant
Jack, Sparrow, manager
John, Dow, programmer
```

# Operator Overloading



## Static method for implementing a certain operator

```
struct Fraction {
    int x, y;
    public Fraction (int x, int y) {this.x = x; this.y = y; }

    public static Fraction operator + (Fraction a, Fraction b) {
        return new Fraction(a.x * b.y + b.x * a.y, a.y * b.y);
    }
}
```

## Usage

```
Fraction a = new Fraction(1, 2);
Fraction b = new Fraction(3, 4);
Fraction c = a + b; // c.x == 10, c.y == 8
```

- The following operators can be overloaded:
  - arithmetic: +, - (unary and binary), \*, /, %, ++, --
  - relational: ==, !=, <, >, <=, >=
  - bit operators: &, |, ^
  - others: !, ~, >>, <<, true, false
- Must always return a function result
- If == (<, <=, true) is overloaded, != (>=, >, false) must be overloaded as well.

# Overloading of && and ||



In order to overload && and ||, one must overload &, |, true and false

```
class TriState {
    int state; // -1 == false, +1 == true, 0 == undecided
    public TriState(int s) { state = s; }

    public static bool operator true (TriState x) { return x.state > 0; }
    public static bool operator false (TriState x) { return x.state < 0; }

    public static TriState operator & (TriState x, TriState y) {
        if (x.state > 0 && y.state > 0) return new TriState(1);
        else if (x.state < 0 || y.state < 0) return new TriState(-1);
        else return new TriState(0);
    }

    public static TriState operator | (TriState x, TriState y) {
        if (x.state > 0 || y.state > 0) return new TriState(1);
        else if (x.state < 0 && y.state < 0) return new TriState(-1);
        else return new TriState(0);
    }
}
```

true and false are called implicitly

```
TriState x, y;
if (x) ...           => if (TriState.true(x)) ...
x = x && y;          => x = TriState.false(x) ? x : TriState.&(x, y);
x = x || y;          => x = TriState.true(x) ? x : TriState.|(x, y)
```

# Conversion Operators



## Implicit conversion

- If the conversion is always possible without loss of precision
- e.g. long = int;

## Explicit conversion

- If a run time check is necessary or truncation is possible
- e.g. int = (int) long;

## Conversion operators for user-defined types

```
class Fraction {
    int x, y;
    ...
    public static implicit operator Fraction (int x) { return new Fraction(x, 1); }
    public static explicit operator int (Fraction f) { return f.x / f.y; }
}
```

## Usage

```
Fraction f = 3;           // implicit conversion, f.x == 3, f.y == 1
int i = (int) f;         // explicit conversion, i == 3
```

# Special Operator Methods

Operators, indexers, properties and events are compiled into “normal” methods

- `get_*` (property getter)
  - `set_*` (property setter)
  - `get_Item` (indexer getter)
  - `set_Item` (indexer setter)
  - `add_*` (event handler addition)
  - `remove_*` (event handler removal)
  - `op_Addition` (binary +)
  - `op_Subtraction` (binary -)
  - `op_Implicit` (implicit cast)
  - `op_Explicit` (explicit cast)
- etc.

# "BCL v2-friendly" Custom Classes 1/3



In order to cooperate smoothly with other BCL classes in the framework 2.0, custom classes should:

- override *ToString* and *GetHashCode*
- overload `==` and `!=`
- implement *ICloneable*

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

```
class MyClass : ICloneable {  
    public object Clone() { return MemberwiseClone(); }  
}
```

# "BCL v2-friendly" Custom Classes 2/3

- implement *IComparable* and *IComparable<T>*

```
public interface IComparable {  
    int CompareTo(object obj); // <0: this < obj, 0: this == obj, >0: this > obj  
}
```

```
public interface IComparable<T> {  
    int CompareTo(T obj); // <0: this < obj, 0: this == obj, >0: this > obj  
}
```

```
class Fraction : IComparable, IComparable<Fraction> {  
    int n, d;  
  
    public int CompareTo(object obj) {  
        if (f == null) return 1;  
        if (!(obj is Fraction)) throw new ArgumentException("Must be of Fraction type.", "obj");  
  
        return CompareTo((Fraction) obj);  
    }  
  
    public int CompareTo(Fraction f) {  
        if (f == null) return 1;  
  
        return n*f.d - f.n*d  
    }  
}
```

# "BCL v2-friendly" Custom Classes 3/3

- override *Equals(object)* and implement *IEquatable<T>*

```
public class Object {  
    public virtual bool Equals(Object obj);  
    ...  
}
```

```
public interface IEquatable<T> {  
    bool Equals(T other);  
}
```

```
class Fraction : IEquatable<Fraction> { // equal to class Fraction : object, IEquatable<Fraction>  
    int n, d;
```

```
    public override bool Equals(object obj) {  
        Fraction f = obj as Fraction;  
        if (f == null) return false;  
        return Equals(f);  
    }
```

```
    public bool Equals(Fraction f) {  
        return f.n == n && f.d == d;  
    }  
}
```

# Structs and Interfaces

```
interface IValue {  
    int Value { get; set; }  
}
```

```
struct MyInt : IValue {  
    private int v;  
  
    public int Value {  
        get { return v; }  
        set { v = value; }  
    }  
  
    public override string ToString() {  
        return v.ToString();  
    }  
}
```

```
class ValueUtils {  
    public static void Set(IValue v, int newValue) {  
        v.Value = newValue;  
        Console.WriteLine("Set: v = {0}", v);  
    }  
}
```

```
class Program {  
  
    static void Main(string[] args) {  
        MyInt mint = new MyInt();  
  
        mint.Value = 10;  
        Console.WriteLine(mint);  
  
        ValueUtils.Set(mint, 20);  
        Console.WriteLine(mint);  
  
        mint.Value = 30;  
        Console.WriteLine(mint);  
    }  
}
```

# Structs and Interfaces

```
interface IValue {  
    int Value { get; set; }  
}
```

```
struct MyInt : IValue {  
    private int v;  
  
    public int Value {  
        get { return v; }  
        set { v = value; }  
    }  
}
```

```
public override string ToString() {  
    return v.ToString();  
}
```

```
class ValueUtils {  
    public static void Set(IValue v, int newValue) {  
        v.Value = newValue;  
        Console.WriteLine("Set: v = {0}", v);  
    }  
}
```

```
class Program {
```

```
    static void Main(string[] args) {  
        MyInt mint = new MyInt();  
  
        mint.Value = 10;  
        Console.WriteLine(mint);  
  
        ValueUtils.Set(mint, 20);  
        Console.WriteLine(mint);  
  
        mint.Value = 30;  
        Console.WriteLine(mint);  
    }  
}
```

C:\Windows\system32\cmd.exe

```
10  
Set: v = 20  
10  
30
```

# Value Types and Interfaces

```
class Program {  
    static bool IsLessThan(Comparable a, Comparable b) {  
        return a.CompareTo(b) < 0;  
    }  
  
    static bool IntIsLessThan(int a, int b) {  
        return (a - b) < 0;  
    }  
  
    static void Main(string[] args) {  
        Console.WriteLine(IsLessThan(10, 20));  
        Console.WriteLine(IntIsLessThan(10, 20));  
    }  
}
```

# IEnumerable and IEnumerator

- following statement:

```
foreach (ElementType element in collection) statement;
```

- is translated into:

```
IEnumerator enumerator = ((IEnumerable) collection).GetEnumerator();  
try {  
    ElementType element;  
    while (enumerator.MoveNext()) {  
        element = (ElementType) enumerator.Current;  
        statement;  
    }  
} finally {  
    IDisposable disposable = enumerator as IDisposable;  
    if (disposable != null) disposable.Dispose();  
}
```

# IEnumerable and IEnumerator (optimized)

- following statement:

```
foreach (ElementType element in collection) statement;
```

- is translated into:

```
var enumerator = collection.GetEnumerator();  
try {  
    ElementType element;  
    while (enumerator.MoveNext()) {  
        element = (ElementType) enumerator.Current;  
        statement;  
    }  
} finally {  
    IDisposable disposable = enumerator as IDisposable;  
    if (disposable != null) disposable.Dispose();  
}
```

# Extension Methods

- Declaration:

```
public static partial class Extensions {  
    public static int ToInt32(this string s) {  
        return Int32.Parse(s);  
    }  
}
```

- Usage:

```
string s = "1234";  
int i = s.ToInt32(); // Same as Extensions.ToInt32(s)
```

- Instance methods take precedence over extension methods
- Extension methods imported in inner namespace declarations take precedence over extension methods imported in outer namespace declarations

# Extension Methods (2)

- Declaration:

```
public static partial class Extensions {  
    public static T[] Slice<T>(this T[] source, int index, int count) {  
        if (index < 0 || count < 0 || source.Length - index < count)  
            throw new ArgumentException();  
        T[] result = new T[count];  
        Array.Copy(source, index, result, 0, count);  
        return result;  
    }  
}
```

- Usage:

```
int[] digits = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};  
int[] a = digits.Slice(4, 3); // Same as Extensions.Slice(digits, 4, 3)
```